NCSC–TG–030
VERSION 1

NATIONAL COMPUTER SECURITY CENTER

A GUIDE TO UNDERSTANDING COVERT CHANNEL ANALYSIS OF TRUSTED SYSTEMS

November 1993

Approved for Public Release:
Distribution Unlimited

NCSC-TG-030
Library No. S-240,572
Version 1

FOREWORD

A Guide to Understanding Covert Channel Analysis of Trusted Systems provides a set of good practices related to covert channel analysis. We have written this guide to help the vendor and evaluator communities understand the requirements for covert channel analysis as described in the *Department of Defense Trusted Computer System Evaluation Criteria* (*TCSEC*). In an effort to provide guidance, we make recommendations in this technical guide that are not cited in the *TCSEC*.

This guide is the latest in a series of technical guidelines published by the National Computer Security Center. These publications provide insight to the *TCSEC* requirements for the computer security vendor and technical evaluator. The goals of the Technical Guideline Program are to discuss each feature of the *TCSEC* in detail and to provide the proper interpretations with specific guidance.

The National Computer Security Center has established an aggressive program to study and implement computer security technology. Our goal is to encourage the widespread availability of trusted computer products for use by any organization desiring better protection of its important data. One way we do this is by supporting the Trusted Product Evaluation Program. This program focuses on the security features of commercially produced and supported computer systems. We evaluate the protection capabilities against the established criteria presented in the *TCSEC*. This program, and an open and cooperative business relationship with the computer and telecommunications industries, will result in the fulfillment of our country's information systems security requirements. We resolve to meet the challenge of identifying trusted computer products suitable for use in processing information that requires protection.

I invite your suggestions for revising this technical guide. We will review this document as the need arises.

Patrick R Gallagher

Patrick R. Gallagher, Jr.
Director
National Computer Security Center

November 1993

ACKNOWLEDGMENTS

The National Computer Security Center (NCSC) extends special recognition and acknowledgment to Virgil D. Gligor as primary author and preparer of this document, to Jonathan K. Millen for providing significant technical input for the covert channel identification and bandwidth estimation sections, and to the first covert channel working group of the NCSC (which met from 1989 to 1991) for providing most of the material presented in Appendices A and B. Capt. James K. Goldston (USAF) and Capt. James A. Muysenberg (USAF) are recognized for the development, editing, and publication of this guide.

We wish to thank the many members of the computer security community who enthusiastically gave their time and technical expertise in reviewing this guide and providing valuable comments and suggestions.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 BACKGROUND

The principal goal of the National Computer Security Center (NCSC) is to encourage the widespread availability of trusted computer systems. In support of this goal, the NCSC created a metric, the *Department of Defense (DoD) Trusted Computer System Evaluation Criteria* (*TCSEC*) [NCSC TCSEC], against which computer systems could be evaluated.

The *TCSEC* was originally published on 15 August 1983 as CSC-STD-001-83. In December 1985, the Department of Defense adopted it, with a few changes, as a Department of Defense Standard, DoD 5200.28-STD. DoD Directive 5200.28, *Security Requirements for Automated Information Systems (AISs)* [DoD Directive], requires the *TCSEC* be used throughout the Department of Defense. The *TCSEC* is the standard used for evaluating the effectiveness of security controls built into DoD AISs.

The *TCSEC* is divided into four divisions: D, C, B, and A. These divisions are ordered in a hierarchical manner, with the highest division (A) being reserved for systems providing the best available level of assurance and security. Within divisions C and B are subdivisions known as classes, which are also ordered in a hierarchical manner to represent different levels of security in these divisions.

1.2 PURPOSE

An important set of *TCSEC* requirements, which appears in classes B2 to A1, is that of covert channel analysis (CCA). The objectives of CCA are:

- Identification of covert channels;

- Determination of covert channels' maximum attainable bandwidth;

- Handling covert channels using a well-defined policy consistent with the *TCSEC* objectives; and

- Generation of assurance evidence to show that all channels are handled according to the policy in force.

To help accomplish these objectives, this guide (1) presents the relative merits of covert channel identification methods and of the covert channel information sources, (2) recommends sound bandwidth determination and handling policies and methods based on the *TCSEC* requirements, and (3) defines the types of evidence that should be provided for handling assurance.

This document provides guidance to vendors on what types of analyses they should carry out for identifying and handling covert channels in their systems, and to system evaluators and accreditors on how to evaluate the manufacturer's analysis evidence. Note, however, that the only measure of *TCSEC* compliance is the *TCSEC*. This guide contains suggestions and recommendations derived from *TCSEC* objectives but which are not required by the *TCSEC*.

This guide is not a tutorial introduction to any topic of CCA. Instead, it is a summary of analysis issues that should be addressed by operating systems designers, evaluators, and accreditors to satisfy the requirements of the B2–A1 classes. Thus, we assume the reader is an operating system designer or evaluator already familiar with the notion of covert channels in operating systems. For this reader, the guide defines a set of baseline requirements and recommendations for the analysis and evaluation of covert channels. For the reader unfamiliar with CCA techniques used to date, the following areas of further documentation and study may be useful:

- Mandatory security models and their interpretation in operating systems [Bell and La Padula76, Biba77, Denning83, Gasser88, Honeywell85a, Honeywell85b, Luckenbaugh86, Rushby85, Walter74];

- Experience with covert channel identification reported in the literature to date [Benzel84, Haigh87, He and Gligor90, Karger and Wray91, Kemmerer83, Lipner75, Loepere85, Millen76, Millen81, Millen89b, Schaefer77, Tsai90, Wray91];

- Bandwidth estimation techniques using standard information theory [Huskamp78, Millen89a, Shannon and Weaver64]; informal bandwidth estimation techniques [Tsai and Gligor88];

- Covert channel handling techniques [Schaefer77, Shieh and Gligor90, Hu91]; and

- Other *TCSEC* guidelines relevant to covert channel handling [NCSC Audit, NCSC Testing].

The reader who is intimately familiar with CCA techniques may want to refer only to the sections on the "*TCSEC* Requirements and Recommendations" (i.e., Sections 3.4, 4.3, and 6.1) and on "Satisfying the *TCSEC* Requirements for Covert Channel Analysis" (Chapter 7).

1.3 SCOPE

This guide refers to covert channel identification and handling methods which help assure that existent covert channels do not compromise a system's secure operation. Although the guide addresses the requirements of systems supporting the *TCSEC* mandatory policy, the analysis and handling methods discussed apply equally well to systems supporting any nondiscretionary (e.g., mandatory) security policy [Saltzer and Schroeder75]. We make additional recommendations which we derive from the stated objectives of the *TCSEC*. Not addressed are covert channels that only security administrators or operators can exploit by using privileged (i.e., trusted) software. We consider use of these channels an irrelevant threat because these administrators, who must be trusted anyway, can usually disclose classified and sensitive information using a variety of other more effective methods.

This guide applies to computer systems and products built with the intention of satisfying *TCSEC* requirements at the B2–A1 levels. Although we do not explicitly address covert channels in networks or distributed database management systems, the issues we discuss in this guide are similar to the ones for those channels.

1.4 CONTROL OBJECTIVE

Covert channel analysis is one of the areas of operational assurance. As such, its control objective is that of assurance. The assurance objective provided in [NCSC TCSEC] is the following:

> Systems that are used to process or handle classified or other sensitive information must be designed to guarantee correct and accurate interpretation of the security policy and must not distort the intent of that policy. Assurance must be provided that correct implementation and operation of the policy exists throughout the system's life-cycle.

This objective affects CCA in two important ways. First, covert channels are the result of an implementation of a nondiscretionary security policy at the operating system level; therefore, depending on how this policy is implemented within a given system, the resulting system will have fewer or more covert channels. Second, the existence of covert channels poses a potential threat to the use of the mandatory policy throughout the system's life cycle. Thus, the identification and handling of covert channels represents an important tenet of mandatory policy support in B2–A1 systems.

1.5 DOCUMENT ORGANIZATION

This guide contains seven chapters, a glossary, a bibliography, and two appendices. Chapter 2 reviews various definitions of covert channels, presents the policy implications of those definitions, and classifies channels. Chapter 3 presents various sources of covert channel information and identification methods, and discusses their relative practical advantages. Chapter 4 describes bandwidth estimation and illustrates a technique based on standard information theory that can be applied effectively in practice. Chapter 5 reviews various covert channel handling methods and policies that are consistent with the *TCSEC* requirements. Chapter 6 discusses covert channel testing and test documentation. Chapter 7 presents *TCSEC* requirements for CCA, and includes additional recommendations corresponding to B2–A1 evaluation classes. The glossary contains the definitions of the significant terms used herein. The bibliography lists the references cited in the text. Appendix A cites some examples of storage and timing channels. Appendix B describes the capabilities of several tools for covert channel identification.

2.0 COVERT CHANNEL DEFINITION AND CLASSIFICATION

In this chapter we provide several definitions of covert channels and discuss the dependency of these channels on implementations of nondiscretionary access control policies (i.e., of policy models). Also, we classify channels using various aspects of their scenarios of use.

2.1 DEFINITION AND IMPLICATIONS

The notion of covert communication was introduced in [Lampson73] and analyzed in [Lipner75, Schaefer77, Huskamp78, Denning83, Kemmerer83], among others. Several definitions for covert channels have been proposed, such as the following:

- *Definition 1* - A communication channel is covert if it is neither designed nor intended to transfer information at all. [Lampson73] (Note: Lampson's definition of covert channels is also presented in [Huskamp78].)

- *Definition 2* - A communication channel is covert (e.g., indirect) if it is based on "transmission by storage into variables that describe resource states." [Schaefer77]

- *Definition 3* - Covert channels "will be defined as those channels that are a result of resource allocation policies and resource management implementation." [Huskamp78] (Note: The computing environment usually carries out resource allocation policies and implementation.)

- *Definition 4* - Covert channels are those that "use entities not normally viewed as data objects to transfer information from one subject to another." [Kemmerer83]

The last three of the above definitions have been used successfully in various security designs for new and retrofitted operating systems and in general covert channel analyses. However, none of the above definitions brings out explicitly the notion that covert channels depend on the type of nondiscretionary access control (e.g., mandatory) policy being used and on the policy's implementation within a system design. A new definition using these concepts can be provided that is consistent with the *TCSEC* definition of covert channels, which states that a covert channel is "a communication channel that allows a process to transfer information in a manner that violates the system's security policy."

- *Definition 5* - Given a nondiscretionary (e.g., mandatory) security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects $I(S_h)$ and $I(S_i)$ of I(M) is covert if and only if any communication between the corresponding subjects S_h and S_i of the model M is illegal in M. [Tsai90]

The above definition has several consequences that help explain the relevance (or lack thereof) of covert channels to different access control policies, as listed below:

(1) Irrelevance of Discretionary Policy Models

The above definition implies that covert channels depend only on the interpretation of nondiscretionary security models. This means the notion of covert channels is irrelevant to discretionary security models.

Discretionary policy models exhibit a vulnerability to Trojan Horse attacks regardless of their interpretation in an operating system [NCSC DAC, Gasser88]. That is, implementations of these models within operating systems cannot determine whether a program acting on behalf of a user may release information on behalf of that user in a legitimate manner. Information release may take place via shared memory objects such as files, directories, messages, and so on. Thus, a Trojan Horse acting on behalf of a user could release user-private information using legitimate operating system requests. Although developers can build various mechanisms within an operating system to restrict the activity of programs (and Trojan Horses) operating on behalf of a user [Karger87], there is no general way, short of implementing nondiscretionary policy models, to restrict the activity of such programs. Thus, given that discretionary models cannot prevent the release of sensitive information through legitimate program activity, it is not meaningful to consider how these programs might release information illicitly by using covert channels.

The vulnerability of discretionary policies to Trojan Horse and virus attacks does not render these policies useless. Discretionary policies provide users a means to protect their data objects from unauthorized access by other users in a relatively benign environment (e.g., an environment free from software containing Trojan Horses and viruses). The role of nondiscretionary policies is to confine the activity of programs containing Trojan Horses and viruses. In this context, the implementation of mandatory policies suggested by the *TCSEC*, which forms an important subclass of nondiscretionary security policies, must address the problem of unauthorized release of information through covert channels.

(2) Dependency on Nondiscretionary Security Policy Models

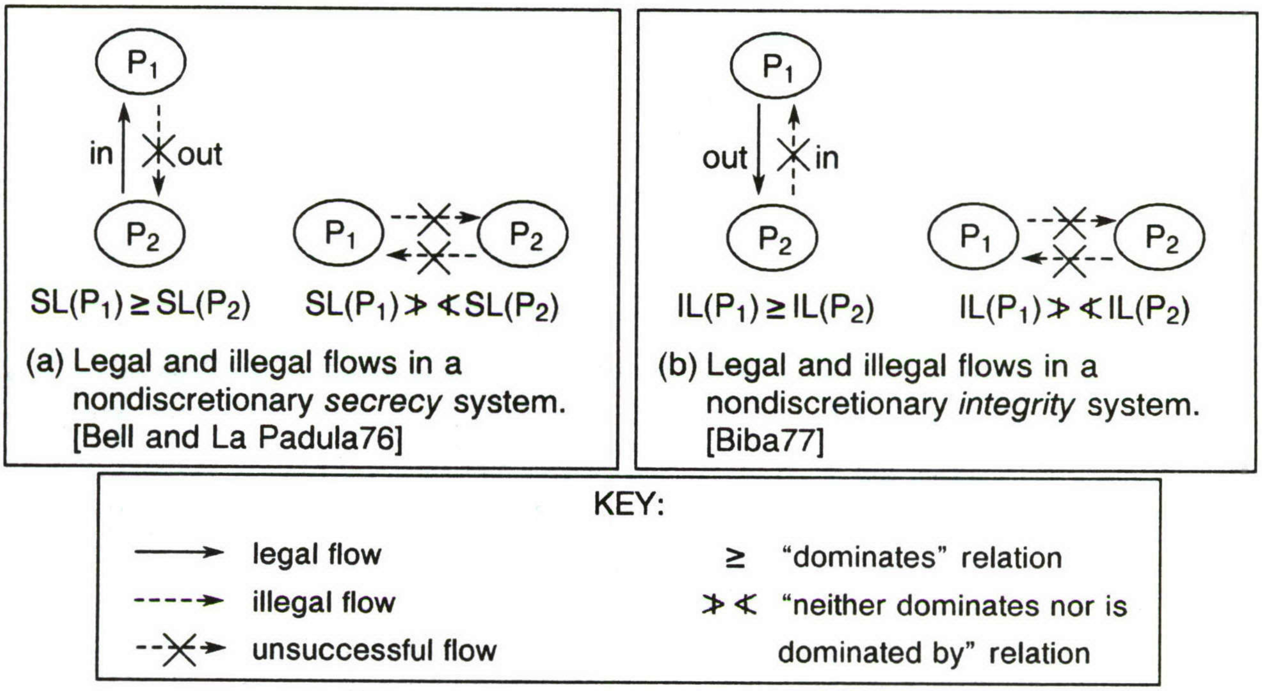
A simple example illustrates the dependency of covert channels on the security policy model used. Consider a (nondiscretionary) separation model M that prohibits any flow of information between two subjects S_h and S_i. Communication in either direction, from S_h to S_i and vice versa, is prohibited. In contrast, consider a multilevel security model, M', where messages from S_h to S_i are allowed only if the security level of S_i dominates that of S_h. Here, some communication between S_h and S_i may be authorized in M'.

The set of covert channels that appears when the operating system implements model M' may be a subset of those that appear when the same operating system implements model M. The covert channels allowing information to flow from S_h to S_i in interpretations of model M could become authorized communication channels in an interpretation of model M'.

The dependency of covert channels on the (nondiscretionary) security policy models does not imply one can eliminate covert channels merely by changing the policy model. Certain covert channels will exist regardless of the type of nondiscretionary access control policy used. However, this dependency becomes important in the identification of covert channels in specifications or code by automated tools. This is the case because exclusive reliance on syntactic analysis that ignores the semantics of the security model implementation cannot avoid false illegal flows. We discuss and illustrate this in sections 3.2.2 and 3.3.

(3) Relevance to Both Secrecy and Integrity Models

In general, the notion of covert channels is relevant to any secrecy or integrity model establishing boundaries meant to prevent information flow. Thus, analysis of covert channels is equally important to the implementation of both nondiscretionary secrecy (e.g., [Bell and La Padula76, Denning76, Denning77, Denning83, NCSC TCSEC]) and integrity models (e.g., [Biba77, Clark and Wilson87]). In systems implementing nondiscretionary secrecy models, such as those implementing the mandatory security policies of the *TCSEC* at levels B2–A1, CCA assures the discovery of (hopefully all) illicit ways to output (leak) information originating from a specific secrecy level (e.g., "confidential/personnel files/") to a lower, or incomparable, secrecy level (e.g., "unclassified/telephone directory/"). Similarly, in systems implementing nondiscretionary integrity models, such analysis also assures the discovery of (hopefully all) illicit ways to input information originating from a specific integrity level (e.g., "valued/personnel registry/") to a higher, or incomparable, integrity level (e.g., "essential/accounts payable/"). Without such assurances, one cannot implement appropriate countermeasures and, therefore, nondiscretionary security claims become questionable at best. Figures 2-1(a) and 2-1(b) illustrate the notion of illegal flows in specific nondiscretionary secrecy and nondiscretionary integrity models.



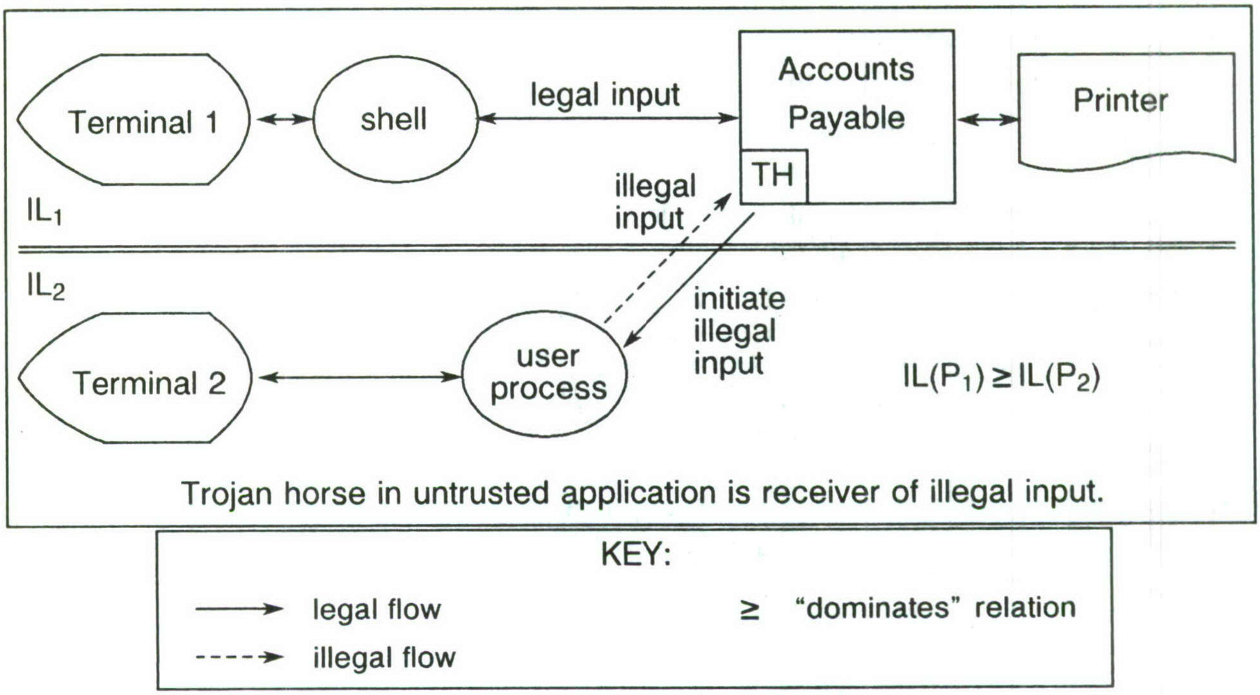
(a) Legal and illegal flows in a nondiscretionary *secrecy* system. [Bell and La Padula76]

(b) Legal and illegal flows in a nondiscretionary *integrity* system. [Biba77]

Figure 2-1. Legal and Illegal Flows

Example 0 – Relevance of Covert Channels to an Integrity Model

Figure 2-2 illustrates the relevance of covert channels to nondiscretionary integrity models. Although this figure assumes a specific nondiscretionary integrity model (i.e., Biba's [Biba77]), covert channels are equally relevant to all nondiscretionary integrity models. In Figure 2-2, a user logged in at the integrity level IL_1 invokes,



Trojan horse in untrusted application is receiver of illegal input.

Figure 2-2. Relevance of Covert Channels to an Integrity Model

through a command processor (i.e., the shell), an accounts payable application that prints payees' names on signed-check papers on a printer. The user is trusted to operate at integrity level IL$_1$ and, by virtue of this trust, his input to the accounts payable application is also classified at integrity level IL$_1$. For similar reasons, both the accounts payable application and the printer are activated at the current integrity level IL$_1$. However, the accounts payable application (and, possibly, the shell) consists of an untrusted set of programs.

The presence of untrusted software in the above example should not be surprising. Most application programs running on trusted computing bases (TCBs) supporting nondiscretionary secrecy consist of untrusted code. Recall that the ability to run untrusted applications on top of TCBs without undue loss of security is one of the major tenets of trusted computer systems. Insisting that all applications that might contain a Trojan Horse, which could use covert channels affecting integrity, be included within an integrity TCB is analogous to insisting that all applications that might contain a Trojan Horse, which could use covert channels affecting secrecy, be included within a secrecy TCB, and would be equally impractical.

If the untrusted accounts payable application contains a Trojan Horse, the Trojan Horse program could send a (legal) message to a user process running at a lower integrity level IL_2, thereby initiating the use of a covert channel. In this covert channel, the Trojan Horse is the receiver of (illegal) lower integrity-level input and the user process is the sender of this input.

The negative effect of exploiting this covert channel is that an untrusted user logged in at a lower integrity level could control the accounts payable application through illegal input, thereby producing checks for questionable reasons. One can find similar examples where covert channels help violate *any* nondiscretionary integrity boundary, not just those provided by lattice-based integrity models (e.g., [Biba77]). Similar examples exist because, just as in the case of TCBs protecting sensitive information classified for secrecy reasons, not all applications running on trusted bases protecting sensitive information for integrity reasons can be verified and proved to be free of miscreant code.

(4) Dependency on TCB Specifications

To illustrate the dependency of covert channels on a system's TCB specifications (Descriptive or Formal Top-Level), we show that changes to the TCB specifications may eliminate existent, or introduce new, covert channels. The specifications of a system's TCB include the specifications of primitives which operate on system subjects, objects, access privileges, and security levels, and of access authorization, object/subject creation/destruction rules, for example. Different interpretations of a security model are illustrated in [Honeywell85a, Honeywell85b, Luckenbaugh86]. Changes to a TCB's specifications may not necessarily require a change of security model or a change of the security model interpretation.

Example 1 – Object Allocation and Deallocation

As an example of the effect of TCB specification changes on covert channel existence (and vice versa), consider the case of an allocator of user-visible objects, such as memory segments. The specifications of the allocator must contain explicit "allocate/deallocate" (TCB) operations that can be invoked dynamically and that subjects can share. A covert channel between the subjects using these user-visible objects exists here [Schaefer77]. However, if the dynamic allocator and, consequently, its specifications are changed to disallow the dynamic allocation/deallocation of objects in a shared memory area, the covert channel disappears. Static object allocation in a shared memory area, or dynamic object allocation in a memory area partitioned on a security level basis, need not change the interpretation of the system's subjects and objects; it only needs to change the specification of the rules for the creation and destruction of a type of object. Although eliminating dynamic sharing of resources and either preallocating objects or partitioning resources on a per-security-level basis represent effective ways to remove some covert channels, they are neither necessary nor possible in all cases because they may cause performance losses.

Though this example illustrates the dependency of covert channels on TCB specifications, it is not a general solution for eliminating covert channels. In fact, we can find other examples to show that changing a TCB's specifications may actually increase the number of covert channels.

Example 2 – Upgraded Directories

As a second example of the strong dependency between the covert channel definition and TCB specifications, consider the creation and destruction of upgraded directories in a system supporting mandatory security and using specifications of interfaces similar to those of UNIX®*. The notion of an upgraded directory [Whitmore73, Schroeder77, Gligor87], its creation and removal, is illustrated in Figures 2-3(a)–(d).

In such a system, whenever a user attempts to remove an upgraded directory from level $L_h > L_i$ where he is authorized to read and write it (as in Figure 2-3(c)), the remove operation fails because it violates the mandatory authorization check (the level of the removing process, L_h, must equal that of the parent directory, L_i). In contrast, the same remove operation invoked by a process at level $L_i < L_h$ succeeds (Figure 2-3(d)).

However, a covert channel appears because of the specification semantics of the remove operation in UNIX "rmdir." This specification says a nonempty directory cannot be removed. Therefore, if the above user logs in at level L_i and tries to remove the upgraded directory from the higher level L_h, the user process can discover whether any files or directories at level $L_h > L_i$ are linked to the upgraded directory. Thus, another process at level L_h can transmit a bit of information to the user

*UNIX is a registered trademark of the UNIX Systems Laboratories.

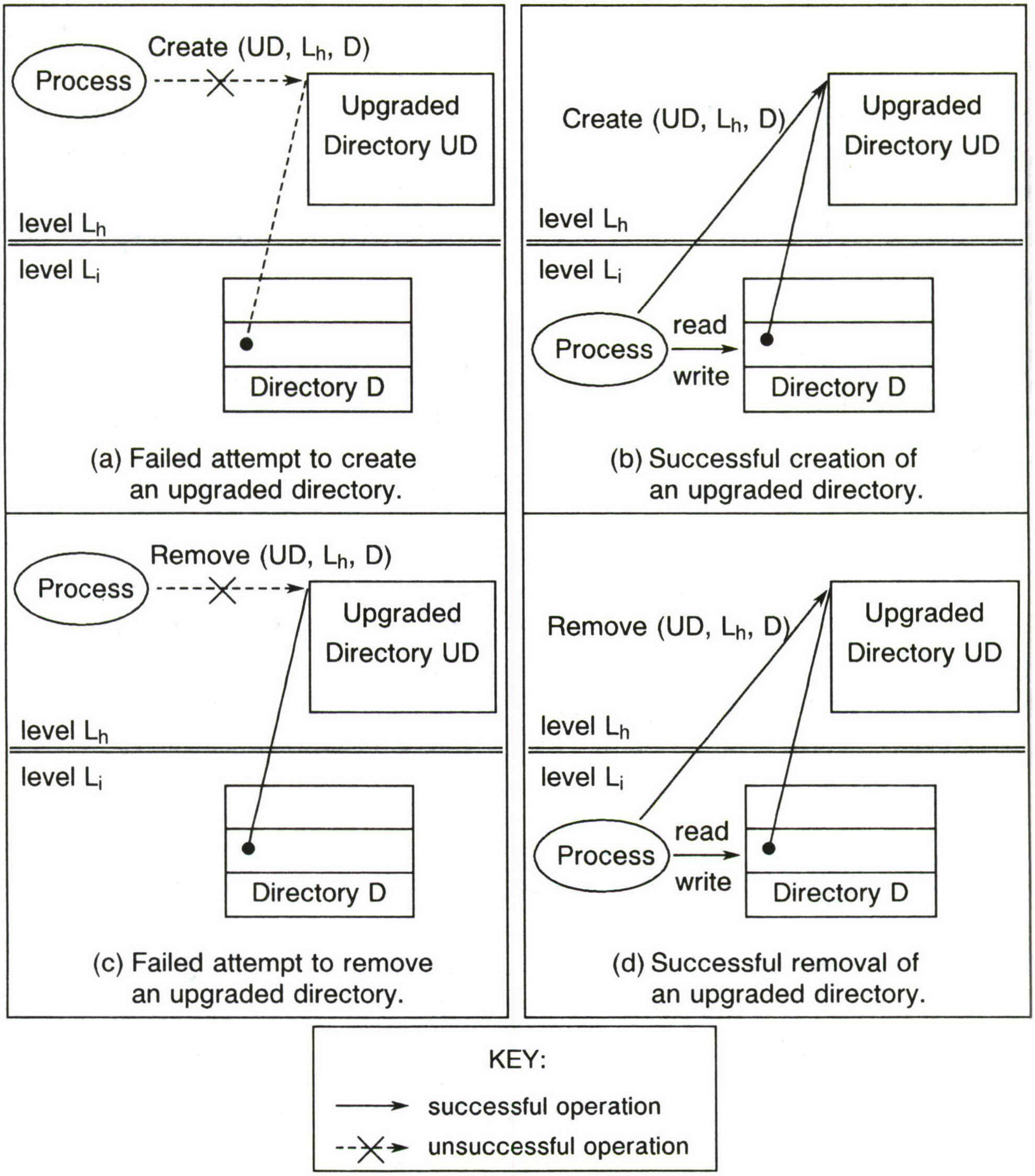


Figure 2-3. Creation and Destruction of an Upgraded Directory at Level $L_h > L_i$

process at level $L_i < L_h$ by creating and removing (e.g., unlinking) files in the upgraded directory. Figure 2-4 illustrates this concept.

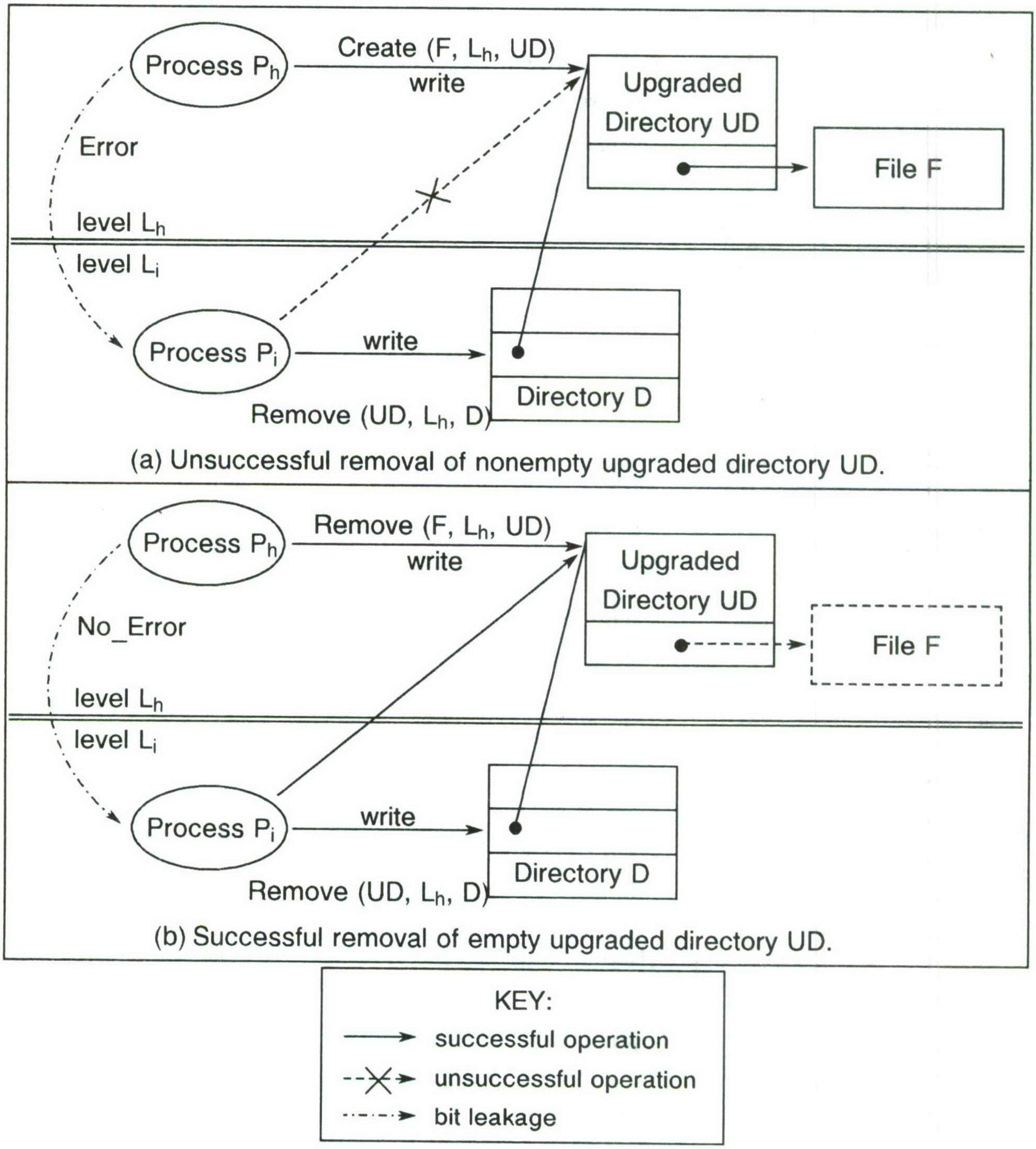


Figure 2-4. Covert Channel Caused by (UNIX) TCB Interface Conventions (where $L_h > L_i$)

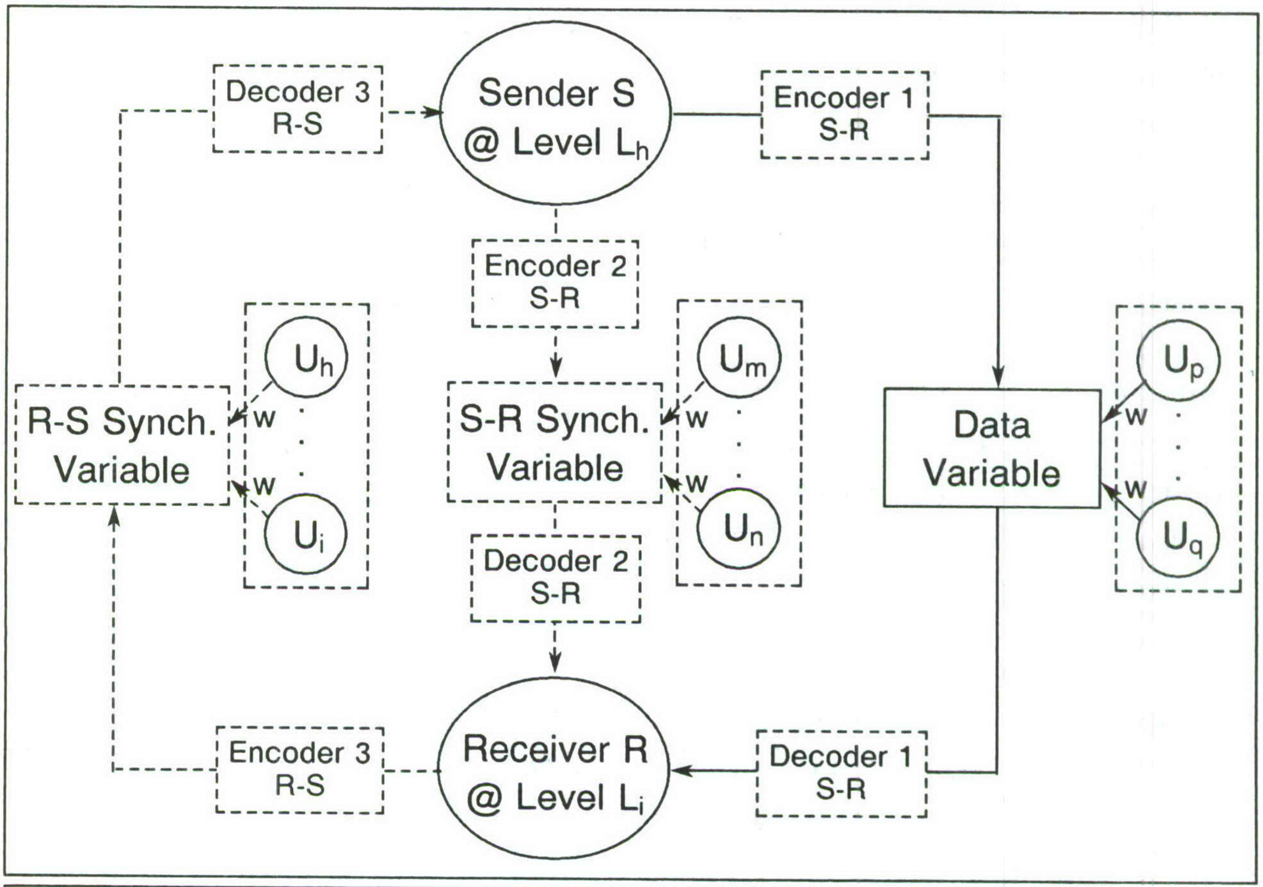
This covert channel would not appear if nonempty directories, and the directory subtree started from them, could be removed (e.g., as in Multics [Whitmore73, Bell and La Padula76]). However, if the specification of directory removal is changed, disallowing removal of nonempty directories (as in UNIX), the covert channel appears. One cannot eliminate the channel without modifying the UNIX user-visible interface. This is an undesirable alternative given that user programs may depend on the interface convention that nonempty UNIX directories cannot be removed. One cannot invent a new TCB specification under which either directories are not user-visible objects or in which the notion of upgraded directories disappears for similar reasons; that is, the UNIX semantics must be modified.

2.2 CLASSIFICATION

2.2.1 Storage and Timing Channels

In practice, when covert channel scenarios of use are constructed, a distinction between covert storage and timing channels [Lipner75, Schaefer77, NCSC TCSEC, Hu91, Wray91] is made even though theoretically no fundamental distinction exists between them. A potential covert channel is a *storage channel* if its scenario of use "involves the direct or indirect writing of a storage location by one process [i.e., a subject of I(M)] and the direct or indirect reading of the storage location by another process." [NCSC TCSEC] A potential covert channel is a *timing channel* if its scenario of use involves a process that "signals information to another by modulating its own use of system resources (e.g., CPU time) in such a way that this manipulation affects the real response time observed by the second process." [NCSC TCSEC] In this guide, we retain the distinction between storage and timing channels exclusively for consistency with the *TCSEC*.

In any scenario of covert channel exploitation, one must define the synchronization relationship between the sender and the receiver of information. Thus, covert channels can also be characterized by the synchronization relationship between the sender and the receiver. In Figure 2-5, the sender and the receiver are asynchronous processes that need to synchronize with each other to send and decode the data. The purpose of synchronization is for one process to notify the other process it has completed reading or writing a data variable. Therefore, a covert channel may include not only a covert data variable but also two synchronization variables, one for sender-receiver synchronization and the other for the receiver-sender synchronization. Any form of synchronous communication requires both the sender-receiver and receiver-sender synchronization either implicitly or explicitly [Haberman72]. Note



Note: Processes $U_h, \ldots, U_i, \ldots, U_m, \ldots, U_n, \ldots, U_p, \ldots, U_q$ may introduce noise whenever they write on the variables, and may introduce delay whenever they do not write on variables and run between the sender and the receiver.

Figure 2-5. Representation of a Covert Channel between Sender S and Receiver R (where $L_h > L_i$ or $L_h \ngtr \nless L_i$)

that synchronization operations transfer information *in both directions*, namely from sender to receiver and vice versa and, therefore, these operations may be indistinguishable from data transfers. Thus, the synchronization and data variables of Figure 2-5 may be indistinguishable.

Some security models, and some of their interpretations, allow receiver-sender communication for subsets of all senders and receivers supported in the system. For example, all mandatory security models implemented in commercial systems to date allow information to flow from a low security level to a higher one. However, sender-receiver synchronization may still need a synchronization variable to inform the receiver of a bit transfer. A channel that does not include sender-receiver synchronization variables in a system allowing the receiver-sender transfer of messages is called a *quasi-synchronous channel*. The idea of quasi-synchronous channels was introduced by Schaefer in 1974 [Reed and Kanodia78].

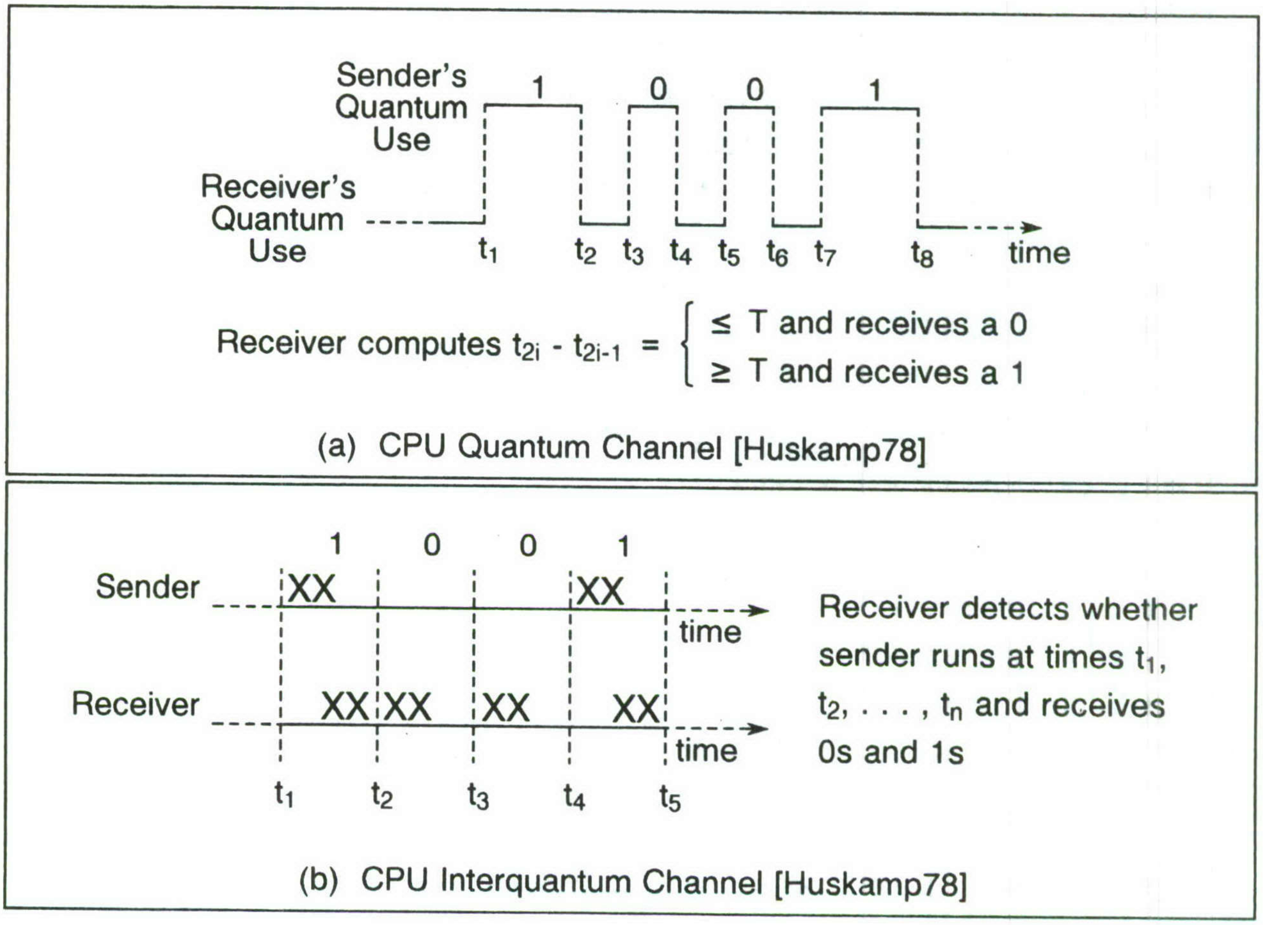
In all patterns of sender-receiver synchronization, synchronization data may be included in the data variable itself at the expense of some bandwidth degradation. Packet-formatting bits in ring and Ethernet local area networks are examples of synchronization data sent along with the information being transmitted. Thus, explicit sender-receiver synchronization through a separate variable may be unnecessary. Systems implementing mandatory security models allow messages to be sent from the receiver to the sender whenever the security level of the sender dominates that of the receiver. In these cases, explicit receiver-sender synchronization through a separate variable may also be unnecessary.

The representation of a covert channel illustrated in Figure 2-5 can also be used to distinguish between scenarios of storage and timing channels. For example, a channel is a storage channel when the synchronization or data transfers between senders and receivers use storage variables, whereas a channel is a timing channel when the synchronization or data transfers between senders and receivers include the use of a common time reference (e.g., a clock). Both storage and timing channels use at least one storage variable for the transmission/sending of the information being transferred. (Note that storage variables used for timing channels may be ephemeral in the sense that the information transferred through them may be lost after it is sensed by a receiver. We discuss this in more detail in Appendix A.) Also, a timing channel may be converted into a storage channel by introducing explicit storage variables for synchronization; and vice versa, a storage channel whose synchronization variables are replaced by observations of a time reference becomes a timing channel.

Based on the above definitions of storage and timing channels, the channels of Examples 1 and 2 are storage channels. Examples 3 and 4 below illustrate scenarios of timing channels. Appendix A presents additional examples of both storage and timing channels.

Example 3 – Two Timing Channels Caused by CPU Scheduling

Quantum-based central processing unit (CPU) scheduling provides two typical examples of timing channels (Figure 2-6). In the first example, the sender of information varies the nonzero CPU time, which it uses during each quantum allocated to it, to send different symbols. For 0 and 1 transmissions, the sender picks two nonzero values for the CPU time used during a quantum, one representing a 0 and the other a 1. This channel is called the "quantum-time channel" in [Huskamp78]. The receiver of the transmitted information decodes the transmitted information by measuring its waiting time for the CPU. If only the receiver and the sender are in the system, the receiver can decode each transmitted bit correctly with probability one for some quantum sizes. A condition of this channel is that the sender be able to block itself before the end of some quantum and reactivate itself before the


Sender's Quantum Use
Receiver's Quantum Use
1
0
0
1
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
time
Receiver computes $t_{2i} - t_{2i-1} = \begin{cases} \leq T \text{ and receives a 0} \\ \geq T \text{ and receives a 1} \end{cases}$

(a) CPU Quantum Channel [Huskamp78]

Receiver detects whether sender runs at times t_1, t_2, . . . , t_n and receives 0s and 1s

(b) CPU Interquantum Channel [Huskamp78]

Figure 2-6. Two CPU Timing Channels

beginning of the next quantum. The sender can meet this condition in a variety of ways depending upon the size of the quantum (e.g., a typical range for quanta is 50–1000 milliseconds). For example, the sender may use an "alarm clock" to put itself to sleep for a fraction of the quantum time, or it may generate a page fault (whose handling may take only a fraction of a quantum time also). A quantum of 100–200 milliseconds is sufficiently large for either case.

In the second example of Figure 2-6, the sender transmits information to the receiver by encoding symbols, say 0s and 1s, in the time between two successive CPU quanta. This channel is called the "interquantum-time channel" [Huskamp78], and is shown in Figure 2-6(b) for the case where only the sender and the receiver appear in the system. To send information, the sender and the receiver agree on set times for sending the information. The transmission strategy is for the sender to execute at time "t_i" if the i-th bit is 1, and to block itself if the i-th bit is 0. The receiver can tell whether the sender executes at time t_i because the receiver cannot execute at the same time.

Example 4 – Other Timing Channels Caused by Shared Hardware Resources

The CPU scheduling channels of Example 3 appear because processes at different secrecy or integrity levels share a hardware resource, namely the CPU. Other sharable hardware resources provide similar timing channels. For example, in any multiprocessor design, hardware resources are shared. Multiple processors share the same bus in shared-bus architectures, share the same memory ports in bus-per-processor architectures, and share multiple busses and memory ports in crossbar-switch architectures, as shown in Figure 2-7. In all multiprocessor architectures, each instruction referencing the memory must lock the shared resource along the CPU-memory interconnection path for at least one memory cycle. (The number of cycles during which the shared resource must be locked depends on the instruction semantics.) Hardware controllers of the shared resource mediate lock conflicts. When the shared resource is no longer needed during the execution of the instruction, the resource is unlocked.

Whenever two processes at two different levels execute concurrently on two separate processors, a covert channel appears that is similar to the CPU interquantum channel presented in Example 3. That is, the sender and the receiver processes establish by prior agreement that the sender process executes at time

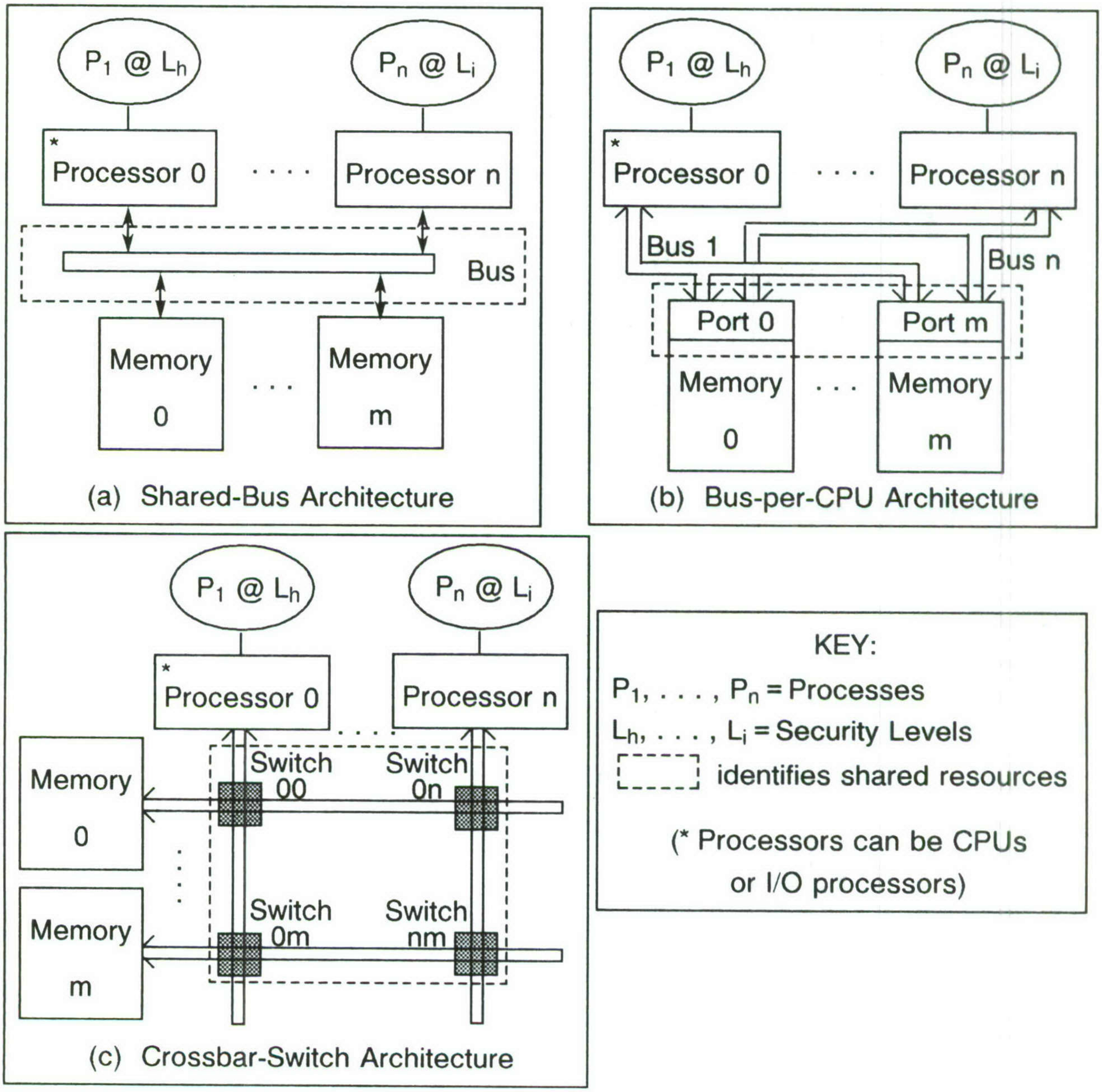


Figure 2-7. Examples of Shared Hardware Resources in Multiprocessor Architectures

"t_i" if the i-th bit is a 1 and does not execute (or at least does not execute memory-referencing instructions) at time "t_i" if the i-th bit is a 0. The receiver can execute a standard set of memory-referencing instructions and time their execution. Thus, the receiver can discover whether the sender executes at time "t_i" by checking whether the duration of the standard set of timed instructions was the expected 1 or longer. As with the CPU channels of Example 3, these channels appear in any multiprocessor system regardless of the nondiscretionary model interpretation. Note that adding per-processor caches, which helps decrease interprocessor contention to shared hardware resources, cannot eliminate these channels. The sender and receiver processes can fill up their caches and continue to exploit interprocessor contention to transmit information.

Appendix A provides other examples of timing channels, which also appear due to the sharing of other hardware resources.

2.2.2 Noisy and Noiseless Channels

As with any communication channel, covert channels can be noisy or noiseless. A channel is said to be noiseless if the symbols transmitted by the sender are the same as those received by the receiver with probability 1. With covert channels, each symbol is usually represented by one bit and, therefore, a covert channel is noiseless if any bit transmitted by a sender is decoded correctly by the receiver with probability 1. That is, regardless of the behavior of other user processes in the system, the receiver is guaranteed to receive each bit transmitted by the sender.

The covert channel of Example 2 is a noiseless covert channel. The sender and receiver can create and remove private upgraded directories, and no other user can affect in any way whether the receiver receives the error/no_error signal. Thus, with probability 1, the receiver can decode the bit value sent by the sender. In contrast, the covert channels of Examples 3 and 4 are noisy channels because, whenever extraneous processes—not just the sender and receiver—use the shared resource, the bits transmitted by the sender may not be received correctly with probability 1 unless appropriate error-correcting codes are used. The error-correcting codes used depend on the frequency of errors produced by the noise introduced by extraneous processes (shown in Figure 2-5) and decrease the maximum channel bandwidth. Thus, although error-correcting codes help change a noisy channel into a noiseless one, the resulting channel will have a lower bandwidth than the similar noise-free channel.

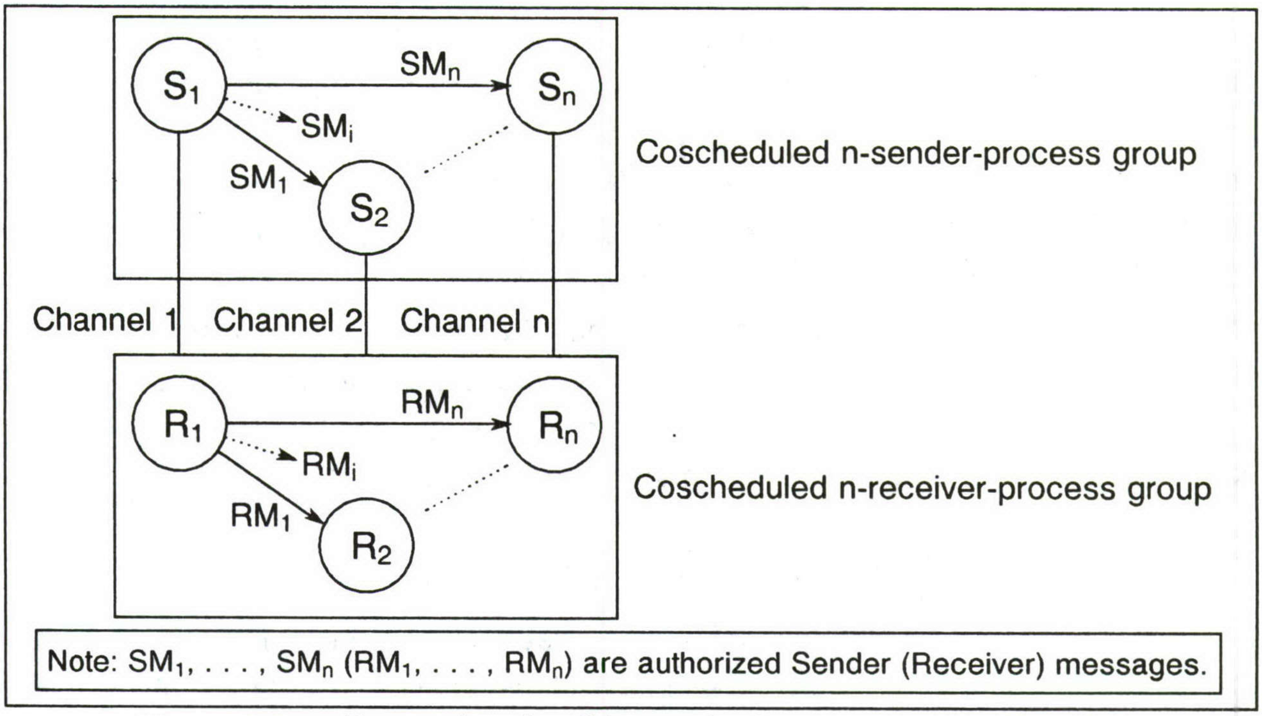
We introduce the term "bandwidth" here to denote the rate at which information is transmitted through a channel. Bandwidth is originally a term used in analog communication, measured in hertz, and related to information rate by the "sampling theorem" (generally attributed to H. Nyquist although the theorem was in fact known before Nyquist used it in communication theory [Haykin83]). Nyquist's sampling theorem says that the information rate in bits (samples) per second is at most twice the bandwidth in hertz of an analog signal created from a square wave. In a covert channel context, bandwidth is given in bits/second rather than hertz, and is commonly used, in an abuse of terminology, as a synonym for information rate. This use of the term "bandwidth" is also related to the notion of "capacity." The capacity of a channel is its maximum possible error-free information rate in bits per second. By using error-correcting codes, one can substantially reduce the error rates of noisy channels. Error-correcting codes decrease the effective (i.e., error-free) information rate relative to the noisy bit rate because they create redundancy in the transmitted bit stream. Note that one may use error-detecting, rather than error-correcting, codes in scenarios where the receiver can signal the sender for retransmissions. All of these notions are standard in information theory [Gallager68].

2.2.3 Aggregated versus Nonaggregated Channels

Synchronization variables or information used by a sender and a receiver may be used for operations on multiple data variables. Multiple data variables, which could be independently used for covert channels, may be used as a group to amortize the cost of synchronization (and, possibly, decoding) information. We say the resulting channels are aggregated. Depending on how the sender and receiver set, read, and reset the data variables, channels can be aggregated serially, in parallel, or in combinations of serial and parallel aggregation to yield optimal (maximum) bandwidth.

If all data variables are set, reset, and read serially, then the channel is serially aggregated. For example, if process P_h of Example 2 (Figure 2-4) uses multiple upgraded directories designated "empty/nonempty" before transferring control to process P_i, the signaling channel will be serially aggregated. Similarly, if all data variables are set, reset, and read in parallel by multiple senders and receivers, then the channel is aggregated in parallel. Note that combinations of serial/parallel aggregation are also possible. For example, the data variables may be set in parallel but read serially and vice versa. However, such combinations do not maximize bandwidth and are, therefore, of limited interest.

Parallel aggregation of covert channel variables requires, for bandwidth maximization reasons, that the sender and receiver pairs be scheduled on different processors at the same time as a group, as illustrated in Figure 2-8 and in [Gligor86]. Otherwise, the bandwidth of the parallel aggregation degrades to that of a serially aggregated channel. The application programmer can strictly control group scheduling of senders and receivers in multiprocessor operating systems such as Medusa or StarOS [Osterhout80, Jones79], which use "coscheduling" [Osterhout82]. Also group scheduling may be possible in multiple workstation systems such as those used in LOCUS [Walker83] or Apollo [Leach83] whenever multiple workstations are available to a single application. In such systems, the analysis of individual covert channels is insufficient to determine the maximum covert channel bandwidth.



Note: $SM_1, \ldots, SM_n$ ($RM_1, \ldots, RM_n$) are authorized Sender (Receiver) messages.

Figure 2-8. Example of n Channels Aggregated in Parallel

Parallel aggregation of covert channels also requires, for bandwidth maximization reasons, that the synchronization messages between all senders, and those between all receivers, be transmitted at a much higher speed than those between senders and receivers. In practice, messages sent among senders, and those sent among receivers, have negligible transmission delays compared to those used by covert channels between senders and receivers. (Also, note that all messages among senders and those among receivers are authorized messages.)

2.3 COVERT CHANNELS AND FLAWED TCB SPECIFICATIONS

An unresolved issue of covert channel definition is whether one can make a distinction between a covert channel and a flaw introduced by the implementation of the security models. In other words, one would like to differentiate between implementation flaws and covert channels, if possible, for practical reasons. For example, both implementors and evaluators of systems supporting mandatory access controls in class B1 could then differentiate between flaws and covert channels. They could determine whether instances of leakage of classified information must be eliminated or otherwise handled or ignored until the B2 level and above.

The covert communication Definition 5 does not differentiate between covert channels and interpretation or TCB specification flaws. This definition implies that, in a fundamental sense, covert channels are in fact flaws of nondiscretionary access control policy implementations, which are sometimes unavoidable in practice regardless of the implementors' design (e.g., Example 3). However, the focus of that definition on the notion of model implementation may help provide a criterion for distinguishing between different types of covert channels or implementation flaws.

To define a distinguishing criterion, let us review Examples 1–4. Examples 1 and 2 show that a change of the TCB specification can, in principle, eliminate the existent covert channels in the specific systems under consideration. In contrast, Examples 3 and 4 show that as long as any system allows the sharing of the CPUs, busses, memory, input/output (I/O) and other hardware resources, covert channels will appear for *any TCB specification*. Furthermore, Example 2 illustrates that, in many systems, a change of TCB specification that would eliminate a covert channel may sometimes be impractical. That is, evidence may exist showing that contemplated changes of the TCB specification would cause a significant loss of compatibility with existing interfaces of a given system. Similar examples can be found to illustrate that changes of TCB specifications may help eliminate other covert channels (or flaws) at the expense of loss of functionality or performance in a given system (e.g., Example 1).

The following criterion may help distinguish between different types of covert channels (or flaws) in practice, thereby providing the necessary input for covert channel, or flaw, handling at levels B1 versus levels B2–A1:

- *Fundamental Channels* - A flaw of a TCB specification that causes covert communication represents a fundamental channel if and only if that flaw appears *under any interpretation* of the nondiscretionary security model in any operating system.

- *Specific TCB Channels* - A flaw of a TCB specification that causes covert communication represents a specific TCB channel if and only if that flaw appears *only under a specific interpretation* of the nondiscretionary security model in a given operating system.

- *Unjustifiable Channels* - A flaw of a TCB specification that causes covert communication represents an unjustifiable channel if and only if that flaw appears *only under a specific but unjustifiable interpretation* of a nondiscretionary security model in a given operating system. (The primary difference between specific TCB and unjustifiable channels is in whether any evidence exists to justify the existence of the respective channels.)

Using this criterion, the covert channels of Examples 3 and 4 are fundamental channels, whereas those of Examples 1 and 2 are specific TCB channels.

The above criterion for distinguishing different types of covert channels (or flaws) suggests the following differentiation policy for B1 and B2–A1 systems. For B1 systems, there should be no handling obligation of fundamental covert channels; specific TCB channels should be handled under the policies in force for classes B2–A1 (as recommended in Chapter 5 of this guide); unjustifiable channels should be eliminated by a change of TCB specification or model implementation for any B-rated systems.

3.0 COVERT CHANNEL IDENTIFICATION

We discuss in this chapter the representation of a covert channel within a system, the sources of information for covert channel identification, and various identification methods that have been used to date and their practical advantages and disadvantages. We also discuss the *TCSEC* requirements for covert channel identification and make additional recommendations.

A covert channel can be represented by a TCB internal variable and two sets of TCB primitives, one for altering (PA_h) and the other for viewing (PV_i) the values of the variable in a way that circumvents the system's mandatory policy. Multiple primitives may be necessary for viewing or altering a variable because, after viewing/altering a variable, the sender and/or the receiver may have to set up the environment for sending/reading the next bit. Therefore, the primary goal of covert channel identification is to discover all TCB internal variables and TCB primitives that can be used to alter or view these variables (i.e., all triples $<variable; PA_h, PV_i>$). A secondary, related goal is to determine the TCB locations within the primitives of a channel where time delays, noise (e.g., randomized table indices and object identifiers, spurious load), and audit code may be placed for decreasing the channel bandwidth and monitoring its use. In addition to TCB primitives and variables implemented by kernel and trusted processes, covert channels may use hardware-processor instructions and user-visible registers. Thus, complete covert channel analysis should take into account a system's underlying hardware architecture, not just kernels and trusted processes.

3.1 SOURCES OF INFORMATION FOR COVERT CHANNEL IDENTIFICATION

The primary sources of information for covert channel identification are:

- System reference manuals containing descriptions of TCB primitives, CPU and I/O processor instructions, their effects on system objects and registers, TCB parameters or instruction fields, and so on;
- The detailed top-level specification (DTLS) for B2–A1 systems, and the Formal top-level specification (FTLS) for A1 systems; and
- TCB source code and processor-instruction (micro) code.

The advantage of using system reference manuals for both TCB-primitive and processor-instruction descriptions is the widespread availability of this information. Every implemented system includes this information for normal everyday use and, thus, no added effort is needed to generate it. However, there are disadvantages to relying on these manuals for covert channel identification. First, whenever system reference manuals are used, one can view the TCB and the processors only as essentially "black boxes." System implementation details are conspicuous by their absence. Thus, using system reference manuals, one may not attain the goal of discovering all, or nearly all, channels. Whenever these manuals are the only sources of information, the channel identification may only rely on guesses and possibly on analogy with specifications of other systems known to contain covert channels. Second, and equally important, is the drawback that analysis based on system reference information takes place too late to be of much help in covert channel handling. Once a system is implemented and the manuals written, the option of eliminating a discovered covert channel by removing a TCB interface convention may no longer be available. Third, few identification methods exist that exhibit any degree of precision and that can rely exclusively on information from system reference manuals. The inadequacy of using only system reference manuals for CCA is illustrated in Example 6 of Section 3.2.3.

Most identification methods developed to date have used formal top-level TCB specifications as the primary source of covert channel identification. The use of top-level specifications has significant advantages. First, these specifications usually contain more detailed, pertinent information than system reference manuals. Second, use of top-level specifications helps detect design flaws that may lead to covert channels in the final implementation. Early detection of design flaws is a useful prerequisite for correct design because one can minimize efforts expended to correct design flaws. Third, tools aiding the identification process exist for the FTLS and thus one gains additional assurance that all channels appearing within the top-level specifications are found (see Appendix B).

However, total reliance on analysis of top-level specifications for the identification of covert channels has two significant disadvantages. First, it cannot lead to the identification of all covert channels that may appear in implementation code. Formal methods for demonstrating the correspondence between information flows of top-level specifications and those of implementation code do not exist to date. Without such methods, guarantees that all covert storage channels in implementation code have been found are questionable at best. The only significant work on specification-to-code correspondence on an implemented system (i.e., the Honeywell SCOMP [Benzel84]) reported in the literature to date has been thorough but informal. This work shows that, in practice, a significant amount of implementation code has no correspondent formal specifications. Such code includes performance monitoring, audit, debugging, and other code, which is considered security-policy irrelevant but which, nevertheless, may contain variables providing potential storage channels.

Second, formal/descriptive top-level specifications of a TCB may not include sufficient specification detail of data structures and code to detect indirect information flows within TCB code that are caused by the semantics of the implementation language (e.g., control statements, such as alternation statements, loops, and so on; pointer assignments, variable aliasing in structures [Schaefer89, Tsai90]). Insufficient detail of specifications used for information flow and storage channel analysis may also cause inadequate implementation of nondiscretionary access controls and channel-handling mechanisms. This is the case because, using the results of top-level specification analysis, one cannot determine with certainty the placement of code for access checks, channel use audits, and time delays to decrease channel bandwidth within TCB code.

In contrast with the significant efforts for the analysis of design specifications, little practical work has been done in applying CCA to implementation code or to hardware. Identifying covert storage channels in source code has the advantages that (1) potentially all covert storage channels can be found (except those caused by hardware), (2) locations within TCB primitives for placement of audit code, delays, and noise can be found, and (3) adequacy of access-check placement within TCB primitives could be assessed [Tsai90]. However, analysis of TCB source code is very labor-intensive, and few tools exist to date to help alleviate the dearth of highly skilled personnel to perform such labor-intensive activity.

3.2 IDENTIFICATION METHODS

All of the widely used methods for covert channel identification are based on the identification of illegal information flows in top-level design specifications and source code, as first defined by [Denning76, 77, 83] and [Millen76]. Subsequent work by [Andrews and Reitman80] on information-flow analysis of programming language statements extended Denning's work to concurrent-program specifications.

3.2.1 Syntactic Information-Flow Analysis

In all flow-analysis methods, one attaches information-flow semantics to each statement of a specification (or implementation) language. For example, a statement such as "a: = b" causes information to flow from b to a (denoted by b→a) whenever b is not a constant. Similarly, a statement such as "if v = k then w: = b else w: = c" causes information to flow from v to w. (Other examples of flows in programming-language statements are found in [Denning83, Andrews and Reitman80, Gasser88]). Furthermore, one defines a flow policy, such as "if information flows from variable x to variable y, the security level of y must dominate that of x." When applied to specification statements or code, the flow policy helps generate flow formulas. For example, the flow formula of "a: = b" is security_level(a) ≥ security_level(b). Flow formulas are generated for complete program and TCB-primitive specifications or code based on conjunctions of all flow formulas of individual language statements on a flow path. (Formula simplifications are also possible and useful but not required.) These flow formulas must be proven correct, usually with the help of a theorem prover. If a program flow formula cannot be proven, the particular flow can lead to a covert channel flow and further analysis is necessary. That is, one must perform semantic analysis to determine (1) whether the unproven flow is real or is a false illegal flow, and (2) whether the unproven flow has a scenario of use (i.e., leads to a real—not just a potential—channel). Example 5 of this section and Examples 7 and 8 of Section 3.3 illustrate the notion of false illegal flow and the distinction between real and potential channels.

Various tools have been built to apply syntactic flow analysis to formal specifications. For example, the SRI Hierarchical Development Methodology (HDM) and Enhanced HDM (EHDM) tools [Feiertag80, Rushby84] apply syntactic analysis to the SPECIAL language. Similarly, the Ina Flo tool of the Formal Development Methodology (FDM) [Eckmann87] and the Gypsy tools [McHugh and Good85, McHugh and Ackers87] have been used for syntactic information-flow analyses. Appendix B reviews these tools. Experience with information-flow analysis in practice is also reported in references [Millen78, Millen81].

Syntactic information-flow analysis has the following advantages when used for covert channel identification:

- It can be automated in a fairly straightforward way;
- It can be applied both to formal top-level specifications and source code;
- It can be applied incrementally to individual functions and TCB primitives; and
- It does not miss any flow that leads to covert channels in the particular specification (or code).

All syntactic information-flow analysis methods share the following three drawbacks:

- Vulnerability to discovery of false illegal flows (and corresponding additional effort to eliminate such flows by manual semantic analysis);
- Inadequacy of use with informal specifications; and
- Inadequacy in providing help with identifying TCB locations for placing covert channel handling code.

All syntactic flow-analysis methods assume each variable or object is either explicitly or implicitly labeled with a specific security level or access class. However, as pointed out in [Kemmerer83], covert channels use variables not normally viewed as data objects. Consequently, these variables cannot necessarily be labeled with a specific security level and, therefore, cannot be part of the interpretation of a given nondiscretionary security model in an operating system. Instead, these variables are internal to kernels or trusted processes and their security levels may vary dynamically depending upon flows between labeled objects. Therefore, the labeling of these variables with specific security levels to discover all illegal flows also renders these code-analysis methods vulnerable to discovery of false flow violations. These false flow violations are called "formal flow violations" in references [Millen78, Schaefer89, Tsai90].

Example 5 – A False Illegal Flow

An example of a false flow violation in the fragment of code shown in Figure 3-1(a) is illustrated in Figures 3-1(b, c). Here, both the alterer and the viewer of the “msgque→mode” variable is the TCB primitive “msgget” of Secure Xenix. The flow

```
SYSTEM_CALL
msgget()
{
    struct a  {
            key_t      key;
            int        msgflg;
    }  *uap;
    register struct msqid_ds *qp  /* ptr to associated q */
    uap = (struct a *)u.u_ap;
    if ((qp = ipcget(uap→msgflg, msgque, &s)) = = NULL)
        return;
    if (obj_access(OBJ_IPCGET, msgque, ASK_READ))
        return;
    u.u_rval1 = qp→msg_perm.seq * v.v_msgmni + (qp - msgque);
}

struct ipc_perm *
ipcget(flag, base, status)
int flag, *status;
register struct ipc_perm *base;
{
    if (base→mode & IPC_ALLOC) {
        u.u_error = ENOSPC;
        return(NULL);
    }
    if (obj_access(OBJ_IPCGET, base, ASK_WRITE))
        return;
    *status = 1;
    base→mode = IPC_ALLOC | (flag & 0777);
    return(base);
}
```

(a) A fictitious fragment of code in a “msgget()” system call.

Figure 3-1. An Example of a False Illegal Flow Caused by Syntactic Flow Analysis

VARIABLE: **msgque→mode**

KEY:
⇒ is an explicit flow [Denning77]
··> is an implicit flow

ALTERER TCB PRIMITIVE: **msgget**

PATH: msgget: (msgflg) ⇒ ipcget: (flag)
msgget: (msgque) ⇒ ipcget: (base)
ipcget: (flag) ⇒ ipcget: (base→mode)

COND: ipcget: !(base→mode & IPC_ALLOC) &&!(obj_access(OBJ_IPCGET, base, ASK_WRITE))

RESULTING FLOW: msgflg ⇒ flag ⇒ msgque→mode

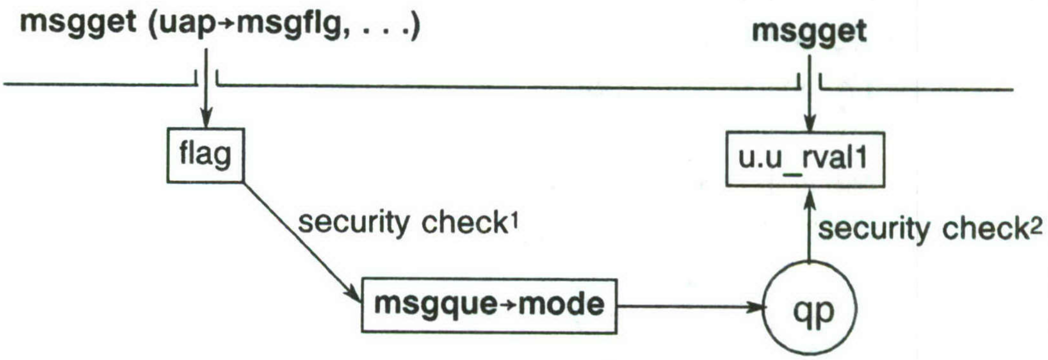
VIEWER TCB PRIMITIVE: **msgget**

PATH: msgget: (msgque) ⇒ ipcget: (base)
ipcget: (base→mode) ··> msgget: (qp)
msgget: (qp) ⇒ msgget: (u.u_rval1)

COND: msgget: !(qp = NULL) &&!(obj_access(OBJ_IPCGET, msgque, ASK_READ))

RESULTING FLOW: msgque→mode ⇒ qp ⇒ u.u_rval1

(b) A flow path and flow condition of code fragment in Figure 3-1(a).



security check[1] = !obj_access(OBJ_IPCGET, base, ASK_WRITE)
security check[2] = !obj_access(OBJ_IPCGET, msgque, ASK_READ)

(c) Graphic representation of the flow path of Figure 3-1(b).

Figure 3-1. An Example of a False Illegal Flow Caused by Syntactic Flow Analysis

formula sl(u.u_rval1) ≥ sl(qp) ≥ sl(msgque→mode) ≥ sl(flag) ≥ sl(uap→msgflg), where sl stands for the security level, cannot be proven because the security levels of the variables vary dynamically, depending on the security levels of the processes invoking the "msgget" primitive. Thus, syntactic flow analysis would identify this flow as illegal. However, an examination of the program conditions under which this flow can actually occur (shown in Figure 3-1(b)) quickly reveals this flow is legal. This flow can occur because the conditions enabling the flow at run time include security checks of the nondiscretionary model interpretations for both viewing and altering InterProcess Communication (IPC) objects. These checks prevent all illegal flows through the "msgque→mode" variable.

Practical examples of false illegal flows appear in all covert channel analyses relying exclusively on syntactic flow analysis. For example, sixty-eight formulas that could not be proven have been found in the SCOMP analysis using the Feiertag Flow tool [Benzel84, Millen89b]. Only fourteen of these flows caused covert channels; the balance were all false illegal flows. Similar examples can be given based on experience with other flow tools. For instance, even in a small (twenty-line) program written in Ina Jo, the Ina Flow tool discovered one hundred-seventeen illegal flows of which all but one were false [Cipher90].

Information-flow analysis does not lend itself to use on informal (e.g., English language) specifications. This means that, if one uses information-flow analysis for B2–B3 class systems, one should apply it to source code. Furthermore, discovery of illegal flows in formal top-level specifications (for class A1 systems) offers little help for identifying TCB locations where covert channel handling code may be necessary. The identification of such locations requires semantic analysis of specifications and code.

3.2.2 Addition of Semantic Components to Information-Flow Analysis

Reference [Tsai90] presents a method for identification of potential storage channels based on (1) the analysis of programming language semantics, code, and data structures used within the kernel, to discover variable alterability/visibility; (2) resolution of aliasing of kernel variables to determine their indirect alterability; and (3) information-flow analysis to determine indirect visibility of kernel variables (e.g., the "msgque→mode" variable in Figure 3-1). These steps precede the application of the nondiscretionary (secrecy or integrity semantic) rules specified in the interpretation of the security model, and implemented in code, to the shared variables and kernel primitives. This last step helps distinguish the real storage channels from the legal or inconsequential ones. The delay in the application of these rules until the security levels of shared variables can be determined with certainty (i.e., from the levels of the objects included in the flows between variables) helps avoid additional (manual) analysis of false illegal flows. Furthermore, discovery of all locations in kernel code where shared variables are altered/viewed allows the correct placement of audit code and time-delay variables for channel-handling mechanisms, and of access checks for nondiscretionary policy implementation.

A disadvantage of this method is that its manual application to real TCBs requires extensive use of highly skilled personnel. For example, its application to the Secure Xenix system required two programmer-years of effort. Thus, using this method in real systems requires extensive use of automated tools. Although the method is applicable to any implementation language and any TCB, its automation requires that different parser and flow generators be built for different languages.

The addition of an automated tool for semantic information-flow analysis to syntactic analysis is reported in [He and Gligor90]. The semantic component of this tool examines all flows visible through a TCB interface and separates the legal from the illegal ones. Since this analysis uses the interpretation of a system's mandatory security model in source code, false illegal flows are not detected. Although one can apply this method to any system, the tool component for semantic analysis may differ from system to system because the interpretation of the mandatory security model in a system's code may differ from system to system. The separation of real covert channels from the potential ones, which requires real scenarios of covert channel use, must still be done manually. Compared to the separation of all potential channels from flows allowing a variable to be viewed/altered through a TCB interface, the separation of real channels from potential channels is not a labor-intensive activity since the number of potential channels is typically several orders of magnitude smaller than the number of flows through a TCB interface.

3.2.3 Shared Resource Matrix (SRM) Method

The SRM method for identifying covert channels was proposed by [Kemmerer83], and used in several projects [Haigh87]. When applied to TCB specifications or code, this method requires the following four steps:

(1) Analyze all TCB primitive operations specified formally or informally, or in source code;

(2) Build a shared resource matrix consisting of user-visible TCB primitives as rows and visible/alterable TCB variables representing attributes of a shared resource as columns; mark each <TCB primitive, variable> entry by R or M depending on whether the attribute is read or modified. (This step assumes one has already determined variable visibility/alterability through the TCB interface.) Variables that can neither be viewed nor altered independently are lumped together and analyzed as a single variable. We show a typical shared-resource matrix in Figure 3-2 and discuss it in Example 6.

(3) Perform a transitive closure on the entries of the shared resource matrix. This step identifies all indirect reading of a variable and adds the corresponding entries to the matrix. A TCB primitive indirectly reads a variable y whenever a variable x, which the TCB primitive can read, can be modified by TCB functions based on a reading of the value of variable y. (Note that whenever the SRM method is applied to informal specifications of a TCB interface as defined in system reference manuals—and not to internal TCB specifications of each primitive, which may be unavailable—performing this step can only identify how processes *outside* the TCB can use information covertly obtained through the TCB interface. Therefore, whenever people using the SRM method treat the TCB as a black box, they can eliminate the transitive closure step since it provides no additional information about flows within the TCB specifications or code.)

(4) Analyze each matrix column containing row entries with either an 'R' or an 'M'; the variable of these columns may support covert communication whenever a process may read a variable which another process can write and the security level of the former process does not dominate that of the

latter. Analysis of the matrix entry leads to four possible conclusions [Kemmerer83]:

(4.1) If a legal channel exists between the two communicating processes (i.e., an authorized channel), this channel is of no consequence; label it "L".

(4.2) If one cannot gain useful information from a channel, label it "N".

(4.3) If the sending and receiving processes are the same, label the channel "S".

(4.4) If a potential channel exists, label it "P".

The labeling of each channel is a useful means of summarizing the results of the analysis.

(5) Discover scenarios of use for potential covert channels by analyzing all entries of the matrix. Examples 7 and 8 of Section 3.2.5 illustrate potential covert channels that cannot be exploited because real scenarios of use cannot be found.

The SRM method has been used successfully on several design specifications [Kemmerer83, Haigh87]. This method has the following advantages:

- It can be applied to both formal and informal specifications of both TCB software and hardware; it can also be applied to TCB source code.
- It does not differentiate between storage and timing channels and, in principle, applies to both types of channels. (However, it offers no specific help for timing channel identification.)
- It does not require that security levels be assigned to internal TCB variables represented in the matrix and, therefore, it eliminates a major source of false illegal flows.

However, lack of security-level assignment to variables has the following negative consequences:

- Individual TCB primitives (or primitive pairs) cannot be proven secure (i.e., free of illegal flows) in isolation. This shortfall adds to the complexity of incremental analysis of new TCB functions.

- The SRM analysis may identify potential channels that could otherwise be eliminated automatically by information-flow analysis.

Although the SRM method is applicable to source code, tools to automate the construction of the shared resource matrix for TCB source code, which is by far the most time-consuming, labor-intensive step, do not exist to date. The manual use of this method on source code—as with other methods applied manually—is susceptible to error.

Example 6 - Inadequacy of Exclusive Reliance on Informal TCB Specifications

The major advantage of the SRM method over syntactic information flow analysis, namely its applicability to informal TCB top-level specifications, is diminished to some extent by the fact that informal top-level specifications lack sufficient detail. We illustrate this observation (1) by showing the results of applying the SRM method to a UNIX TCB specification as found in the Xenix reference manuals [IBM87] using three internal variables of the file subsystem (i.e., “mode,” “lock,” and “file_table”) as the target of CCA, and (2) by comparing this analysis with the automated analysis performed for the same three variables and the Secure Xenix TCB source code with the tool presented in [He and Gligor90].

Figure 3-2 illustrates the results of this comparison. In this figure, the bold-faced matrix entries denote the information added to the original SRM matrix as a result of the automated analysis. This figure shows that about half of the relevant primitives were missed for one of the variables (i.e., “mode”) and a third were missed for another variable (i.e., “file_table”).

Furthermore, more than half of the R/M entries constructed for the primitives found to reference the three variables in system manuals were added R/M designators by the automated analysis of source code. Although different results can be obtained by applying the SRM method to different informal specifications, this example illustrates that the application of SRM (and of any other method) to informal specification can be only as good as the specifications.

PRIMITIVES	SHARED GLOBAL VARIABLES		
	mode	lock	file table
access	*R*		
chmod	**R** *M*		
chsize	**R**	**M**	
close	**R**	**M**	**R** *M*
creat	*R M*	**R M**	*R M*
dup	**R**	**M**	**R**
exec	**R M**	**M**	**R**
fcntl		*R M*	**R M**
fstat	**R M**		
link	**R M**		
locking	**R**	*R M*	
open	*R M*	**R M**	*R M*
read	**R M**		
stat	**R M**		
unlink	**R** *M*		
utime	**R M**		
write	**R M**	**M**	

(1) R = read and M = modify.

(2) Primitives in **Boldface**: primitives that are found to be able to read/modify a shared global variable in the source code but not in the specification (DTLS).

(3) Primitives in *Italics*: primitives that are found to be able to read/modify a shared global variable both in the source code and in the specification (DTLS).

(4) Rules (2) and (3) also apply to the readability/modifiability of primitives to a shared global variable.

PRIMITIVES	SHARED GLOBAL VARIABLES		
	mode	lock	file table
aclcreat			**R M**
aclopen			**R M**
brk	**R M**		
brkctl	**R M**		
chdir	**R M**		
chown	**R M**		
creatsem	**R M**		**R M**
exit	**R M**		**R**
fork	**R M**		**R**
ioctl	**R**		
mknod	**R M**		
nbwaitsem	**R**		
opensem	**R M**		**R M**
pipe	**R M**		**R M**
rdchk	**R**		
sdfree	**R M**		
sdget	**R M**		
seek	**R**		
shmat	**R M**		
shmctl	**R M**		
shmget	**R M**		
sigsem	**R**		
sync	**R M**		
vhangup			**R**
waitsem	**R**		

Figure 3-2. Shared Resource Matrix for Three Variables

3.2.4 Noninterference Analysis

Noninterference analysis of a TCB requires one to view the TCB as an abstract machine. From the point of view of a user process, a TCB provides certain services when requested. A process' requests represent the abstract machine's inputs, the TCB responses (e.g., data values, error messages, or positive acknowledgements) are its outputs, and the contents of the TCB internal variables constitute its current state. Each input results in a (TCB) state change (if necessary) and an output. Each input comes from some particular process running at a particular security level, and each output is delivered only to the process that entered the input that prompted it.

[Goguen and Meseguer82] formulated the first general definition of information transmission in the state-machine view of a TCB, generalizing on an earlier but more restricted definition by [Feiertag80]. They defined the concept of noninterference between two user processes. The definition was phrased in terms of an assumed initial or start-up state for the machine. It stated, in effect, that one user process was noninterfering with another when the output observed by the second user process would be unchanged if all inputs from the first user process, ever since the initial state, were eliminated as though they had never been entered. Goguen and Meseguer reasoned that if inputs from one user process could not affect the outputs of another, then no information could be transmitted from the first to the second. (One can verify this property using Shannon's definition of information transmission [Millen 89b].)

To define noninterference precisely, let X and Y be two user processes of a certain abstract-machine TCB. If w is a sequence of inputs to the machine, ending with an input from Y, let Y(w) be the output Y receives from that last input (assuming the machine was in its initial state when w was entered). To express noninterference, w/X is the subsequence that remains of w when all X-inputs are deleted, or "purged," from it. Then X is noninterfering with Y if, for all possible input sequences w ending with a Y-input, Y(w) = Y(w/X).

It is somewhat unintuitive that noninterference relates a whole sequence of inputs, including, perhaps, many X-inputs, to a single Y-output. In CCA, the traditional view is that whenever a covert channel exists between X and Y, each individual X-input has an effect on the next Y-output. Noninterference analysis suggests another view may be appropriate, however. Note that user process Y might enter an input to request an output at any time. Suppose, in fact, that Y enters an input every time X did. Ignoring other inputs, the overall input sequences looks like: $x_1y_1x_2y_2 \ldots x_ny_n$. The definition of noninterference applies not only to the whole sequence, but to all the initial segments of it ending in a Y-input, namely: (x_1y_1), $(x_1y_1x_2y_2)$, . . . , $(x_1y_1 \ldots x_ny_n)$. Noninterference requires that *every* Y output is unaffected by *all* previous X inputs. Thus, it seems necessary to analyze all past X inputs because of the following: Suppose each X input is reported as a Y output after some delay; a covert channel arises just as it would if the X input came out immediately in the next Y output.

In practice, it is cumbersome to analyze the entire history of inputs to the machine since its initial state. However, this analysis is unnecessary because the current state has all the information needed to determine the next Y-output. Thus, noninterference of X with Y can be expressed in terms of the current state instead of the whole prior input history.

Clearly, if X is noninterfering with Y, an X input should have no effect on the next Y output. Noninterference is actually stronger than this, however, since it requires that an X input has no effect on *any subsequent* Y output. To avoid analyzing unbounded input sequences, it is useful to partition TCB states into equivalence classes that are not distinguishable using present or subsequent Y outputs. That is, two states are Y-equivalent if (1) they have the same Y output in response to the same Y input, and (2) the corresponding next states after any input are also Y-equivalent. (This definition is recursive rather than circular; this is computer science!) [Goguen and Meseguer84] proved a theorem, called the "Unwinding Theorem," which states that X is noninterfering with Y if and only if each X input takes each state to a Y-equivalent state; a simpler version of this theorem was given by [Rushby85].

Unwinding is important because it leads to practical ways of checking noninterference, especially when given a formal specification of a TCB that shows its states and state transitions. The multilevel security policy requires that each process X at a given security level should interfere only with a process Y of an equal or higher security level. To apply this requirement in practice, the TCB states must be defined, and the Y-equivalent states must be determined.

A straightforward way of identifying Y-equivalent states in a multilevel secure TCB is to label state variables with security levels. If Y is cleared for a security level s, then the two states are Y-equivalent if they have the same values in those state variables having a security level dominated by s. A less formal way of expressing this statement is that Y has (or should have) a blind spot when it tries to observe the current state. Y can observe state variables at or below its own level, but state variables at a higher level are in the blind spot and are invisible. So two states are Y-equivalent if they look the same under Y's "blind spot" handicap.

The state-variable level assignment must have the property that the effect of any input turns equivalent states into equivalent states. This means that invisible variables cannot affect the visible part of the state. This property is one of three that must be proved in a noninterference analysis. The other two properties are that (1) any return values reported back to Y depend only on variables visible to Y, and (2) an input from a higher level user process X cannot affect the variables visible to user process Y.

Noninterference analysis has the following important advantages:

- It can be applied both to formal TCB specifications and to source code;
- It avoids discovery of false illegal flows; and
- It can be applied incrementally to individual TCB functions and primitives.

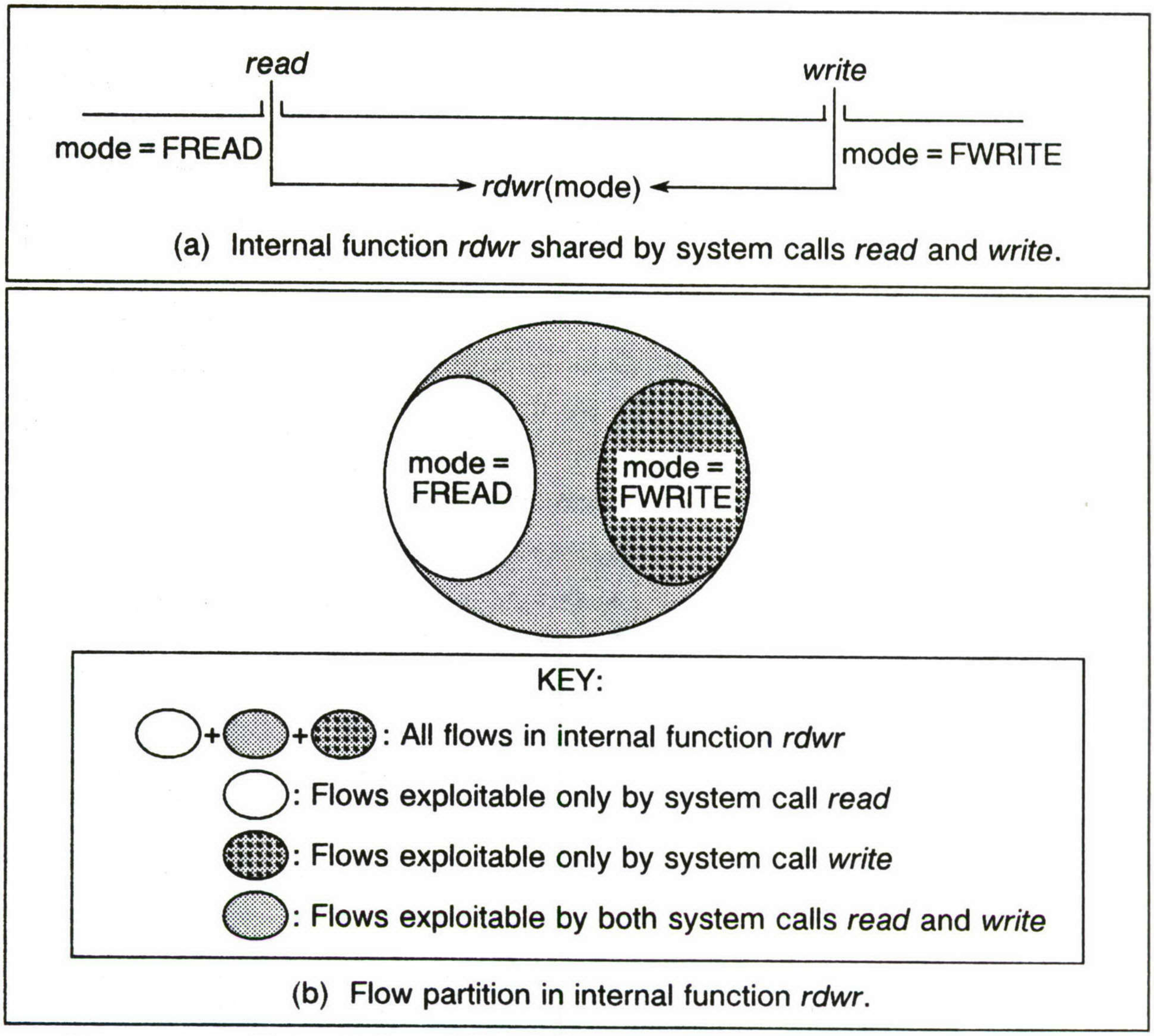
However, it has three practical disadvantages. First, one can only apply it to formal TCB top-level specifications and, possibly, to source code. Therefore, its application to systems in classes where analyses of formal specifications or source code is not required (i.e., class B2–B3 systems) can only be recommended but not mandated. Only the A1 system design, which requires specification-to-code correspondence, can be construed to require covert channel identification on source code (during the specification-to-code correspondence). Second, manual application of noninterference to significant-size TCBs may be impractical, and automated tools are currently unavailable for use in significant-size systems. Third, noninterference analysis is "optimistic." That is, it tries to prove that interference does not appear in TCB specifications or code. Thus, its best application is TCB specifications of trusted-process isolation rather than to TCB components containing large numbers of shared variables (i.e., kernels). Noninterference analysis was used to discover covert channels of the Logical Co-processing Kernel (LOCK)—a successor of the Secure Ada Target (SAT) [Boebert85]. The process of using the Gypsy system to verify noninterference objectives, and the consequences of discovering that a real operating system does not quite attain them, was discussed in reference [Haigh87].

3.3 POTENTIAL VERSUS REAL COVERT CHANNELS

Covert channel identification methods applied statically to top-level specifications or to code produce a list of *potential* covert channels. Some of the potential covert channels do not have scenarios of real use. These potential channels are artifacts of the identification methods. However, false illegal flows do not necessarily cause these potential channels. As illustrated in Figure 3-1 (b), all flows have a condition that enables the flow to take place as the system runs (e.g., dynamically). A general reason why a potential covert channel may not necessarily be a real covert channel is that, at run time, some flow conditions may never become true and, thus, may never enable the illegal flow that could create a covert channel. Another reason is that the alteration (viewing) of a covert channel variable may not be consistent with the required alteration (viewing) scenario. For example, a field of the variable may be altered but it could not be used in the scenario of the covert channel. Similarly, not all TCB primitives of a channel can be used in real covert channel scenarios. The ability to use some TCB primitives of a channel to transfer information may depend on the choice of the primitive's parameters and the TCB state. Examples 7, 8, and 9 illustrate these cases. To determine whether a potential covert channel is a real covert channel, one must find a real-time scenario enabling an illegal flow.

Example 7 - An Example of a Potential Covert Channel

Figure 3-3(a) illustrates the difference between potential and real covert channels. Two UNIX TCB primitives "read" and "write" share the same internal function "rdwr" but pass different values to the parameter of this function. CCA on the internal function "rdwr" reveals all possible information flows within "rdwr" (i.e., both flows that lead to real channels and flows that only lead to potential channels). Among the latter are flows with the condition "mode = FWRITE." These flows cannot be exploited by TCB primitive "read" because it can never enable this condition. Similarly, TCB primitive "write" cannot exploit those flows with the condition "mode = FREAD." Thus, among the potential covert channels arising from the invocation of the internal function "rdwr," those with condition "mode = FWRITE" cannot


read
write
mode = FREAD
mode = FWRITE
rdwr(mode)
mode = FREAD
mode = FWRITE
KEY:
○+▒+▨ : All flows in internal function rdwr
○: Flows exploitable only by system call read
▨: Flows exploitable only by system call write
▒: Flows exploitable by both system calls read and write

(a) Internal function *rdwr* shared by system calls *read* and *write*.

(b) Flow partition in internal function *rdwr*.

Figure 3-3. Potential and Real Covert Channels Corresponding to Different Flow Partitions

be real covert channels for the "read" primitive, and those with the condition "mode = FREAD" cannot be real covert channels for the "write" primitive. Real-time scenarios do not exist for those potential covert channels. Figure 3-3 (b) shows the partitioning of flows in internal function "rdwr" based on whether a flow can be exploited by the "read" and "write" primitives.

Example 8 – Real and Potential Covert Channels in Secure Xenix

Figure 3-4 illustrates examples of real and potential covert channels of Secure Xenix. The two tables shown in Figure 3-4 contain two basic types of covert storage

Channel TCB Var Primitive	File Table	Inode Table	Disk Space	Message ID Table	Process ID Table	Text Table
creat	AV	AV	AV			
exec			AV			AV
fork	a	a	AV		AV	
msgget				AV		
msgctl				A		
open	AV	AV	AV			
wait					A	

(a) Examples of resource-exhaustion channels in Secure Xenix™

Channel TCB Var Primitive	Number of Free Blocks	Number of Free Inodes	Message Identifier	Process Identifier
creat	A	A		
chsize	A	a		
fork	A	a		AV
msgget			V	
msgctl			A	
ustat	V	V		

(b) Examples of event-count channels in Secure Xenix™

Figure 3-4. *Examples of Potential and Real Channels*

channels: resource-exhaustion and event-count channels. Resource-exhaustion channels arise wherever system resources are shared among users at more than one security level. To use a resource-exhaustion channel, the sending process chooses whether or not to exhaust a system resource to encode a signal of a 0 or 1. The receiving process detects the signal by trying to allocate the same system resource. Depending on whether the resource can be allocated, the receiving process can determine the value of the signal from the sending process.

In event-count channels, the sending process encodes a signal by modifying the status of a shared system resource (but not exhausting the resource). By querying the status of the resource, either through TCB primitives returning the resource status explicitly or by observing the return result of some TCB primitives that allocate the resource, the receiving process can detect the signal from the sending process.

In the tables of Figure 3-4, each row is marked with a TCB primitive and each column with a shared global variable. Entries in the tables indicate whether a TCB primitive can alter (A or a) and/or view (V or v) the corresponding global variable. An upper-case A in an entry indicates that the TCB primitive can alter the global variable as the means of encoding and transferring information through the variable. Similarly, an upper-case V in an entry indicates that the TCB primitive can view the value of the global variable and detect the signal transmitted by sending user processes. Thus, for any shared global variable, the set of TCB primitives that have a capital A and those that have a capital V constitute a real covert channel. For example, TCB primitives "creat" and "open" can be used by the sending and the receiving processes to transfer information through the shared global variable representing the "file_table" in the system. On the other hand, a lower-case a in an entry means that, although the TCB primitive can alter the global variable, the alteration cannot be used for encoding information in the global variable. For example, the execution of the TCB primitive "fork" alters the "file_table" because it increments the file reference count for the child process it creates. This alteration, however, is different from that of allocating a file-table entry and, thus, it does not provide a real scenario for sending a signal. Similar reasoning explains why the entries marked with a lowercase v in Figure 3-4 cannot be used in real scenarios of covert channels.

The distinction between an alteration of a global variable that can be used to encode information (i.e., entry denoted by A) and one that cannot (i.e., entry denoted by a) can be eliminated if a finer partitioning of the "file_table" structure is performed. That is, if the file reference count of the "file_table" is viewed as a separate variable within the "file_table" structure, then the TCB primitive "fork" would not appear to alter the "file_table" structure. Instead, "fork" would alter only the new variable, namely, the file reference count. In either case, however, the covert channel analysis should yield the same results.

Example 9 – Dependencies on TCB State and Primitive Parameters

The covert channel examples of Figure 3-5 illustrate both system-state and parameter dependencies found in UNIX systems. For example, the primitive "creat"

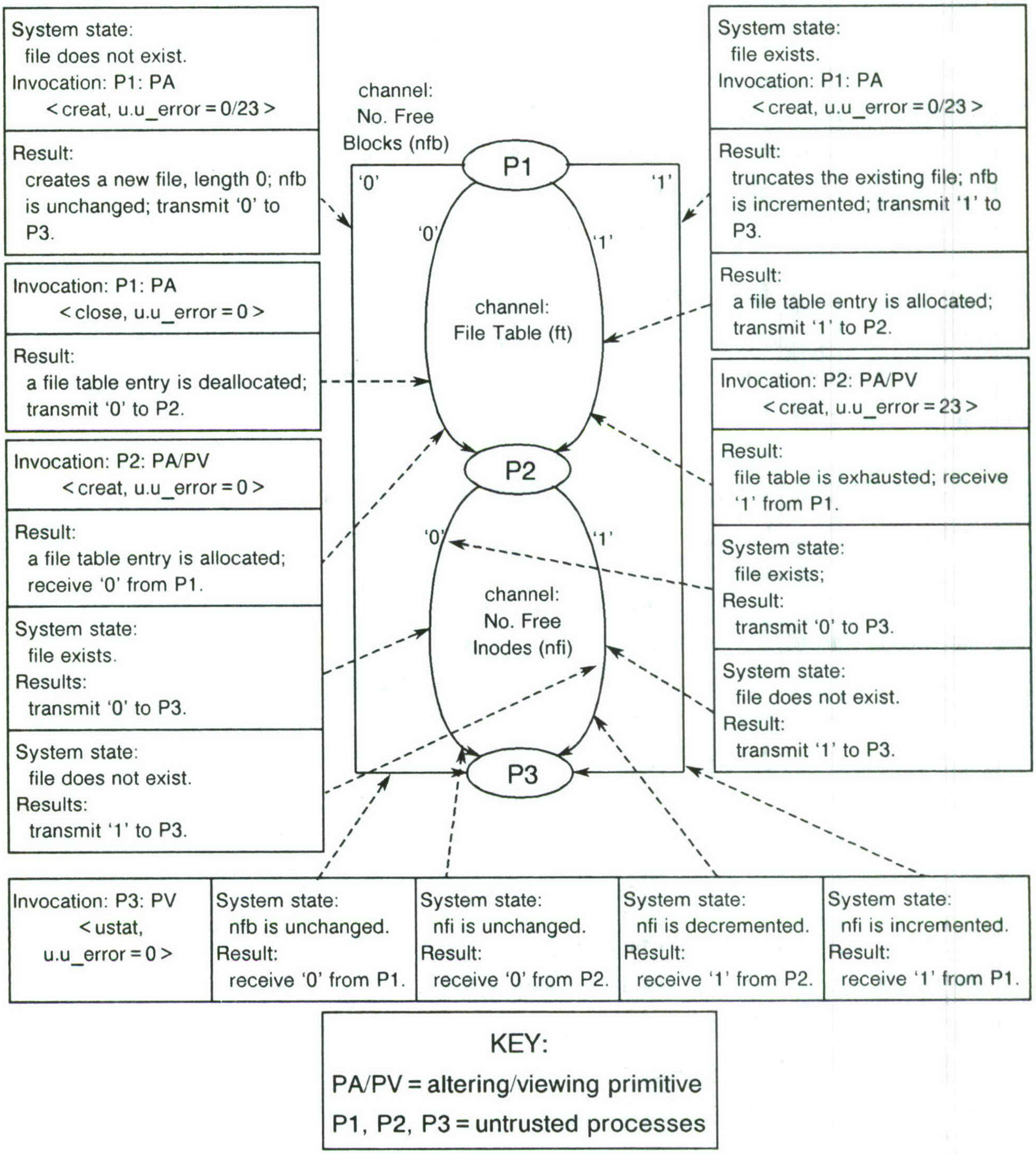


Figure 3-5. An Example of the Use of Multiple Channels by Three Processes

can alter (i.e., decrement) the total number of free inodes (nfi) only if the object to be created does not exist. If the object exists, "creat" has no effect on nfi. In addition, the primitive "creat" can be used to alter (i.e., increment) the total number of free blocks (nfb) in the system if the file being created currently exists. That is, if the file exists, "creat" truncates the file, and as a result increments nfb. Otherwise, "creat" has no effect on nfb. (The disk-block-space channel is also affected by this condition.) Furthermore, the alteration of the disk-block-space channel, and of the nfi and nfb channels by the primitive "creat," is determined by the file system specified in the parameter of the "creat" invocation.

The example of Figure 3-5 also illustrates the combined state and parameter dependencies. Consider again the channel that modulates the nfb and the disk-block-space channel. Primitive "chsize" can be used to alter these channel variables (i.e., deallocate memory and increase the total number of free blocks) only if the file on which it is applied exists, and only if its parameter indicates file shrinkage. When used to expand the size of an existing file, primitive "chsize" does not alter the channel variables but merely changes the ip-i_size field of the inode.

Other examples of parameter dependency and combined state and parameter dependencies, unrelated to those of Figure 3-5, can be found. For example, the primitive "semget(key, nsems, semflg)" can affect the semaphore-identifier channel and the semaphore-map exhaustion channel. Within this primitive, if parameter "key" is equal to IPC_CREAT, thereby denoting the creation of a semaphore, a semaphore identifier, its associated semaphore data structure, and a set containing "nsems" semaphores are created for key. In contrast, if parameter key is not equal to IPC_CREAT, nothing is created.

Furthermore, if parameter key does not already have a semaphore identifier associated with it, and if the Boolean expression (semflg and IPC_CREAT) is true, a "semget" call creates for parameter key a semaphore identifier, its associated data structure, and the set containing "nsems" semaphores. If parameter key already has a semaphore identifier associated with it, a new semaphore structure is not created.

3.4 *TCSEC* REQUIREMENTS AND RECOMMENDATIONS

Covert channel identification requirements appear for the classes B2–A1 of the [NCSC TCSEC]. The B2 requirements of CCA state that the "system developer shall conduct a thorough search for storage channels . . . ."

For class B2, the search for covert storage channels should be conducted on system reference manuals and on the DTLS of the TCB. Although the *TCSEC* does not require storage channel identification in the TCB source code and in hardware (microcode) specifications, such a search would ensure the completeness of the identification results. Although no specific identification method is required, arbitrary, ad hoc, undocumented methods, which system evaluators cannot repeat on independently selected test cases, are unacceptable. This nonacceptance is justified by the notion that system developers must conduct a thorough search for covert channels and an evaluation team must evaluate the search.

Use of any identification method on informal top-level specifications may yield incomplete results, as illustrated in Example 6 of this section in the context of the SRM method. For this reason, it seems important to apply the storage channel identification method to the TCB source code. Otherwise, the thoroughness of the covert channel identification search may be in doubt. Furthermore, source-code analysis may be useful in the definition of covert channel scenarios that help distinguish real and potential channels. Therefore, we recommend analyzing both the DTLS and the source code for covert channel identification at class B2.

The B3-class requirement of the *TCSEC* extends the above B2-class requirement to all covert channels (i.e., to timing channels). Although this extension imposes no added requirement in terms of the identification method used, timing channel scenarios should be developed. These scenarios should include all system sources of independent timing such as the CPU and the I/O processors. Inclusion of all these sources will provide additional assurance that classes of timing channels are not overlooked. [Huskamp78] provides an example of complete timing channel analysis for the CPU scheduling channels.

The A1-class requirement of CCA includes all the B2–B3-class requirements and extends them by stating, “Formal methods shall be used in analysis.”

One may apply CCA methods to both formal specifications and source code of the TCB. Examples of such methods include syntactic information-flow analysis (with or without the use of semantic analysis), SRM, and noninterference analysis. Other formal methods for covert channel identification may exist and may be equally suitable at level A1. The identification method chosen by the developer should be applied to the FTLS. Unless the identification of covert channels is made a part of the specification-to-code correspondence, in which case source-code analysis is included, we recommend complementing FTLS analysis with formal or informal source-code analysis. Otherwise, covert channels may remain undetected.

4.0 COVERT CHANNEL BANDWIDTH ESTIMATION

In this chapter we discuss various factors that affect the covert channel bandwidth computation, including TCB primitive selection, parameter and state dependencies, and channel aggregation. We also present both information-theory-based and informal methods for maximum bandwidth estimation, and discuss various factors that degrade the covert channel bandwidth. The *TCSEC* requirements and recommendations are also discussed.

4.1 FACTORS AFFECTING THE BANDWIDTH COMPUTATION

The computation of covert channel bandwidths is one of the key aspects of covert channel analysis. This is the case because most decisions about how to handle identified channels rely on the determination of channel bandwidth. Therefore, it is important to examine briefly the factors that primarily affect the computation of covert channel bandwidth.

4.1.1 Noise and Delay

Two of the most important factors affecting the bandwidth of a covert channel in any operating system or hardware platform are the presence of noise and the delays experienced by senders and receivers using the channel. The primary sources of noise and delay are the processes $U_p, \ldots, U_q$ shown in Figure 2-5, which can interpose themselves due to scheduling of various hardware resources between senders and receivers. Although these processes can degrade the maximum attainable bandwidth of a channel significantly (e.g., up to about 75% [Tsai and Gligor88]), the degradation is not certain in all architectures and at all times since it depends on the nature of the multiprogrammed system (e.g., single user, multiprocess workstation, multiuser time sharing system) and on the system load. Thus, while the noise and delay factors are significant, the computation of the maximum attainable bandwidth of any channel must discount both noise and delays, and must assume that only the senders and receivers are present in the system [Millen89a].

4.1.2 Coding and Symbol Distribution

In general, the attainable maximum bandwidth (i.e., capacity) depends on the choice of symbol encoding scheme agreed upon by a sender and a receiver. Coding schemes exist that allow the exploitation of the maximum attainable bandwidth of a channel on the distribution of symbols in the space of transmitted messages [Millen89a]. However, informal covert channel analysis usually assumes a 0 or a 1 represents each symbol transmitted. Thus, the distribution of 0s and 1s becomes an important factor of bandwidth computation whenever using informal methods. Where the time required to transmit a 0 is close (on the average) to the time required to transmit a 1, one can assume that 0s and 1s are used with approximately equal frequency. This assumption is based on the fact that the bandwidth (i.e., capacity) of discrete memoryless channels is maximized under such distributions. (Section 4.2 below illustrates both an informal bandwidth-computation method, where such distributions are assumed, and an information-theory-based method, where such distributions are not assumed.)

Informal bandwidth computation methods do not achieve, in general, the maximum bandwidth of a channel because they do not use appropriate coding techniques. Formal bandwidth-computation methods not only allow the precise determination of attainable maximum bandwidth but also help define coding schemes that can be used to attain those bandwidths [Millen89a].

4.1.3 TCB Primitive Selection

In most systems, covert channel identification associates multiple TCB primitives with a covert channel variable. For example, most UNIX covert channel variables can be altered or viewed by a number of primitives that varies between about ten and forty. Among the primitives of each variable, one must select those having the highest speed for the bandwidth computation. Although one should measure each primitive's speed with only senders and receivers using the system, one should not conduct these measurements independently of the covert channel scenario of use (i.e., without using parameters and TCB state conditions that would be present when a channel is in use). Otherwise, the bandwidth computation could lead to unrealistically high or low values. Low values may cause security exposures whereas high values may cause performance degradation whenever delays are used based on bandwidth values. We can illustrate the latter case by the "chsize" primitive of UNIX. The speed of this primitive depends on whether a file is shrunk (low speed) or expanded (higher speed). However, the use of "chsize" with the expand option cannot be made in covert channels requiring disk free block alteration because this primitive does not alter the disk free block variable. We discuss this in more detail in the next section.

4.1.4 Measurements and Scenarios of Use

The performance measurements of the TCB primitives of covert channels require one to include not only the altering and viewing primitives but also the performance of the primitives that initialize the environment for the altering and viewing of a variable. The environment initialization primitives may differ for the altering and viewing primitives. For example, the environment initialization primitive for altering a variable to transmit a 1 may differ from that necessary to transmit a 0. Similarly, the environment initialization primitives for viewing a 1 may differ from those necessary for viewing a 0. Furthermore, the same primitive may use different amounts of time depending upon whether it is used to set (read) a zero or a one (e.g., whether it returns an error). Scenarios of covert channel use are needed to determine which environment initialization primitives must be taken into account. Section 4.2 provides examples of different environment initialization primitives and their use for two real covert channels of UNIX.

Also included in the measurements is the process- or context-switching time. The measurement of this time is needed because, during the transmission of every bit of information through a covert channel, control is transferred between the sender and receiver at least twice. In most operating systems, the measurement of the minimum process switching time is a fairly complex task. The reason for this complexity is that with every process switch the measurement environment changes and, therefore, the measurement of each switch may yield different values. Sometimes it is also difficult to measure individual process-switching times because process switching may be possible only as a side-effect of some primitives. Other processes may be scheduled to run during a switch from one process to another, thereby adding unwarranted delay to switching time. To eliminate the difference between measured process-switching times within the same system, one must ensure that only a few processes are present in the system when taking measurements and repeat the measurements a large number of times (e.g., a hundred thousand times) to ensure choosing the minimum value.

Real scenarios of covert channel use include sender-receiver synchronization. This synchronization delays the covert channel and, therefore, decreases the channel's bandwidth. However, since one cannot predict the synchronization scenario (because it is privately agreed upon by the sender and receiver), we generally assume the bandwidth decrease caused by synchronization is negligible. This assumption helps ensure computing the maximum bandwidth.

All primitive measurements and process-switching time measurements must be repeatable. Otherwise, independent evaluators cannot verify the bandwidth computations.

4.1.5 System Configuration and Initialization Dependencies

TCB primitive measurements and process switching times depend very heavily on a number of system architecture parameters. These parameters include:

- System-component speed (e.g., disk, memory, and CPU);
- System configuration (e.g., configurations using or not using caches);
- Configuration-component sizes (e.g., memory sizes, cache sizes); and
- Configuration initialization.

The least obvious dependency is memory size. The same measurement on two systems configured identically but using different memory sizes may yield different results. For example, in systems with smaller memory the primitives will appear to be slower due to the additional swapping and buffer management necessary to accommodate the measurement environment. Similarly, to ensure repeatable results, one must properly initialize the measurement environment.

4.1.6 Aggregation of Covert Channels

In general, both serial and parallel aggregation of distinct covert channels can increase the effective bandwidth available to senders and receivers for covert transmission of information. The easiest way to approximate the effect of aggregation on the maximum channel bandwidths is to (1) set the context-switching time to zero for both serial and parallel aggregation, and (2) to sum the bandwidths of the individual channels for parallel aggregation. However, the *TCSEC* requirements and guidelines neither require nor recommend that one consider aggregation in covert channel analysis. However, one needs to consider the notion of channel aggregation in the area of threat analysis, whenever such analysis is performed in the environment of system use (see Section 5.4 below).

4.1.7 Transient Covert Channels

Transient covert channels are those which transfer a fixed amount of data and then cease to exist. Normally, bandwidth and capacity calculations apply only to channels that are sustainable indefinitely. Thus, it would seem transient channels are an irrelevant threat. However, if a large volume of data can be leaked through a transient channel, one must consider channel bandwidth analysis and handling, for the threat of channel use becomes real.

4.2 BANDWIDTH ESTIMATION METHODS

4.2.1 Information-Theory-Based Method for Channel-Bandwidth Estimation

Millen presents in [1989a] a method based on Shannon's information theory [Shannon and Weaver64]. In this method, one assumes the covert channels are noiseless, no processes other than the sender and receiver are present in the system during channel operation, and the sender-receiver synchronization takes a negligible amount of time. These assumptions are justified if the goal is the computation of the maximum attainable bandwidth. With these assumptions, one can model most covert channels that arise in practice as finite-state machines (graphs). Furthermore, these graphs are deterministic in that for any state transition corresponding to a given channel symbol (e.g., 0 or 1), only one next state appears in the graph. Figure 4-1 illustrates a state graph for a two-state channel. Most covert channels of interest in practice can be represented with two-state graphs. This is because, for most

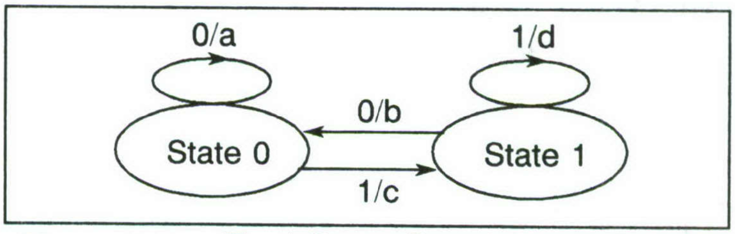


Figure 4-1. Two-State Graph for a Covert Channel

channels, the current state of the channel depends on the last signal sent and, thus, only two states are necessary to capture the scenario of information transfer.

Example 10 – Scenario for a Two-State Covert Channel

A scenario for the transfer of 0s and 1s using a two-state graph can be defined by associating each transition of the graph with the transfer of a 0 or a 1. Each transition covers a sender's action followed by a receiver's action.

For example, to send 0 in state 0:

- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 0;
- receiver resets the environment (if necessary);
- receiver transfers control to sender.

To send 0 in state 1:

- sender sets the channel variable to 0;
- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 0;
- receiver resets the channel environment (if necessary);
- receiver transfers control to sender

To send 1 in state 0:

- sender sets the channel variable to 1;
- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 1;
- receiver transfers control to sender

To send 1 in state 1:

- sender transfers control to receiver;
- receiver reads the channel variable;
- receiver records 1;

- receiver transfers control to sender.

One can determine the time required to send a 0 or 1 by listing the corresponding sequence of TCB primitive calls and adding up their times. Recall that the TCB primitive calls, and their total duration, depend on the state of the channel. Also, recall that the reading (not just the setting) of a 0 or a 1 will have different durations even if they are represented by the same TCB primitive call. For example, if the reading of a 0 in one state is represented by the "open" primitive call with a successful return and in the other state by "open" with a failure return, the reading of the 0 in the two states will have different durations because the latter call always has a shorter duration. The sequences of TCB primitive calls necessary to transfer 0s and 1s using a two-state graph may be different, and thus they may take four different amounts of time, say a, b, c, and d time units, respectively (as shown in Figure 4-1).

To determine the bandwidth of a channel represented with a two-state graph, N(t), one must find the number of possible transmissions of duration t. The bandwidth (i.e., capacity) of a channel can be expressed in terms of $N_h(t)$ as follows:

$C = \lim(\log_2 N_h(t))/t.$

To find $N_h(t)$, let $N_0(t)$ be the total possible number of transmissions of duration exactly t beginning in one of the two states, and let $N_1(t)$ be the total possible number of transmissions of duration exactly t beginning in the other state. (In general, there will be an $N_h(t)$ for the h-th state where h ranges over the state set.) The number of transmissions satisfies a system of difference equations that can be read off the two-state graph. Each equation is based on the fact that the set of transmissions beginning in a given state consists of a union of several disjoint sets, discriminated by the initial symbol of the transmissions. The number of transmissions with a given initial symbol is equal to the total number of (shorter) transmissions beginning in the next state after the transition for that symbol.

The following system of equations can be used for the file-lock channel:

$N_0(t) = N_0(t - a) + N_1(t - c)$
$N_1(t) = N_0(t - b) + N_1(t - d)$

In general, the h-th equation has the form:

$$N_h(t' = \sum_i i(N_i(t - T_{hi}),$$

where T_{hi} is the time taken by a transition from state h to state i.

Note that $N_0(t)$ is nonzero only for those values of t that are expressible as a sum of multiples of a, b, c, and d. To determine the bandwidth of the channel, it is only necessary to find the asymptotic upper limit of $N_0(t)$ as t approaches infinity [Shannon and Weaver64]. This may be found in the form:

$$N_h(t) = A_h{}^*x^t$$

Substituting this solution, we obtain the system of equations:

$$A_h{}^*x^t = \sum_i i(A_i{}^*x^{t-T_{hi}})$$

and $C = \lim(\log_2(A_h(x^t))/t = \log_2 x$, when t→infinity.

Note that there may be multiple solutions for x in the above equations. The largest solution provides the bandwidth (capacity).

We can express this system of equations in matrix form as $(P\text{-}I)A = 0$, where P is a matrix of negative powers of x. Since (P-I) is singular, its determinant Det(P-I) = 0. Figure 4-2 shows the system of equations, their determinant, and the solution.

Example 11 – Application to Two Secure Xenix Channels

Two of the Secure Xenix channels whose bandwidths were computed in reference [Tsai and Gligor88] for a PC/AT configuration are the inode table channel and the upgraded directory channel. In this example we illustrate Millen's method described above using measurements of Secure Xenix TCB primitives on an IBM PS/2 model 80 configuration. T_{cs} represents the context switch time, which is 3 milliseconds. The values of T_r (T_s) represent the duration of reading (setting) the covert channel variable, and the value of T_{env} represents the duration of setting up the transfer environment (e.g., a state transition).

$$\begin{cases} A_0x^{t} = A_0x^{t-a} + A_1x^{t-c} \\ A_1x^{t} = A_0x^{t-b} + A_1x^{t-d} \end{cases}$$

$$\Rightarrow \begin{cases} (x^{-a} - 1)A_0x^{t} + x^{-c} \; A_1x^{t} = 0 \\ x^{-b} \; A_0x^{t} + (x^{-d} - 1)A_1x^{t} = 0 \end{cases}$$

$$\Rightarrow \begin{vmatrix} x^{-a} - 1 & x^{-c} \\ x^{-b} & x^{-d} - 1 \end{vmatrix} = 0$$

$$\Rightarrow x^{-(a+d)} - x^{-a} - x^{-d} + 1 - x^{-(b+c)} = 0$$

Thus,

$$C = \lim_{t->infinity} (\log_2 N_h(t))/t \text{ implies:}$$

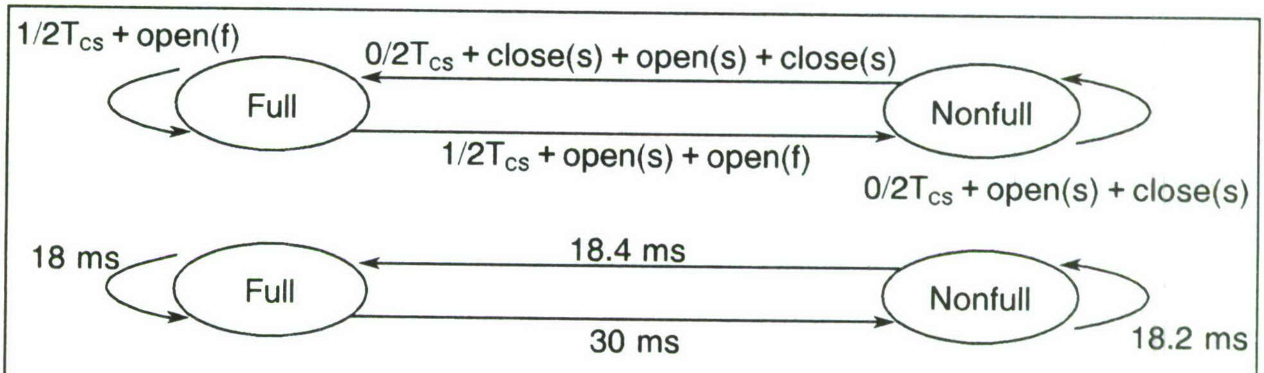
$$C = \lim_{t->infinity} (\log_2 A_h x^{t})/t = \log_2 x$$

where $N_h(t)$ is the total number of transmissions of duration exactly t in state h.

Figure 4-2. Simultaneous Equations, Determinant, Capacity (Bandwidth)

The Inode Table Channel

In this example the state 0 of the inode table channel is represented by the inode table full state, and the state 1 by the inode table nonfull state. Figure 4-3 shows the state transitions defined below and their durations.



1/2T_cs + open(f)
0/2T_cs + close(s) + open(s) + close(s)
Full
Nonfull
1/2T_cs + open(s) + open(f)
0/2T_cs + open(s) + close(s)
18 ms
18.4 ms
Full
Nonfull
30 ms
18.2 ms

$$x^{-36.2} - x^{-18} - x^{-18.2} + 1 - x^{-48.4} = 0$$

$$x = 1.003356346 \qquad 0 < \log_2 x < 1/18$$

$$C = 1000 \log_2 x = 47.63 \text{ bits/sec}$$

Figure 4-3. State Graphs for the Inode Table Channel

State 0:

When the inode table is full, two T_{cs} and one viewing primitive "open(f)" with a failure return are needed to transfer a 1 from a sending process to a receiving process. Thus, the following times are needed to transfer a 1 from state 0:

T_r(full→full) = open(f), T_S(full→full) = 0, T_{env}(full→full) = 0.

When switching from the full state to the nonfull state, an alteration primitive "close(s)," a viewing primitive "open(s)," an environment set-up primitive "close(s)," and two T_{cs} are needed to send a 0. Thus, the following times are needed to transfer a 0 from state 0:

T_r(full→nonfull) = open(s), T_S(full→nonfull) = close(s), T_{env}(full→nonfull) = close(s).

State 1:

When the transition is from the nonfull state to the nonfull state, a viewing primitive "open(s)," an environment set-up primitive "close(s)," and two T_{cs} are needed to transfer a 0. Thus, the following times are needed to transfer a 0 from state 1:

T_r(nonfull→nonfull) = open(s), T_s(nonfull→nonfull) = 0, T_{env}(nonfull→nonfull) = close(s).

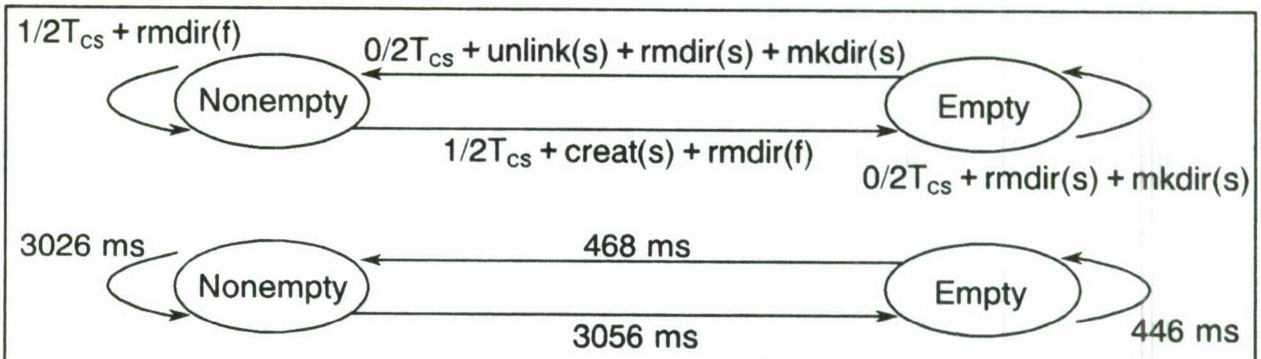
When switching from the nonfull state to the full state, an alteration primitive "open(s)," a viewing primitive "open(f)," and two T_{cs} are needed to transfer a 1. Thus, the following times are needed to transfer a 0 from state 1:

T_r(nonfull→full) = open(f), T_S(nonfull→full) = open(s), T_{env}(nonfull→full) = 0.

The bandwidth (i.e., capacity) of this channel, denoted by C in Figure 4-3, is 47.63 bits/second.

The Upgraded Directory Channel

In this example the state 0 of the upgraded directory channel is represented by the directory-full state, and the state 1 by the directory-nonfull state. The state transitions defined below and their durations are shown in Figure 4-4.



$$x^{-3472} - x^{-3026} - x^{-446} + 1 - x^{-3524} = 0$$

$$x = 1.00051598 \qquad 0 < \log_2 x < 1/446$$

$$C = 1000 \log_2 x = 0.512 \text{ bits/sec}$$

Figure 4-4. *State Graphs for the Upgraded Directory Channel*

State 0:

When an upgraded directory is nonempty, two T_{cs} and one viewing primitive "rmdir(f)" with a failure return are needed to transfer a 1 from a sending process to a receiving process.

T_r(nonempty→nonempty) = rmdir(f), T_s(nonempty→nonempty) = 0,
T_{env}(nonempty→nonempty) = 0.

When switching from the nonempty state to the empty state, an alteration primitive "unlink(s)," a viewing primitive "rmdir(s)," an environment set-up primitive "mkdir(s)," and two T_{cs} are needed to send a 0. Thus, the following times are needed to transfer a 0 from state 0:

T_r(nonempty→empty) = rmdir(s), T_s(nonempty→empty) = unlink(s),
T_{env}(nonempty→empty) = mkdir(s).

State 1:

When the transition is from the empty state to the empty state, a viewing primitive "rmdir(s)," an environment set-up primitive "mkdir(s)," and two T_{cs} are needed to transfer a 0. Thus, the following times are needed to transfer a 0 from state 1:

T_r(empty→empty) = rmdir(s), T_s(empty→empty) = 0, T_{env}(empty→empty) = mkdir(s).

When switching from the empty state to the nonempty state, an alteration primitive "creat(s)", a viewing primitive "rmdir(f)", and two T_{cs} are needed to transfer a 1. Thus, the following times are needed to transfer a 1 from state 1:

T_r(empty→nonempty) = open(f), T_s(empty→nonempty) = open(s),
T_{env}(empty→nonempty) = 0.

The bandwidth of this channel is denoted by C in Figure 4-4 and is 0.512 bits/second.

4.2.2 Informal Method for Estimating Covert Channel Bandwidth

A simple formula for computing the maximum attainable bandwidth of a noiseless covert channel in absence of any spurious processes that would delay senders and receivers was presented in [Tsai and Gligor88]. The formula is:

$B(0) = b^*(T_r + T_s + 2T_{cs})^{**}(-1)$,

In this formula, b represents the encoding factor (which we assume to be 1 in most practical cases), and

$$T_r = \sum_{i=1}^{n} \frac{T_r(i) + T_{env}(i)}{n}$$

$$T_s = \sum_{i=1}^{n} \frac{T_s(i)}{n} ,$$

where n is the number of total possible transitions. $T_s(i)$ and $T_r(i)$ are the times necessary to set and read a 0 or a 1 after having transmitted a 0 or a 1. Thus, n = 4. $T_{env}(i)$ is the time to set up the environment to read a 0 or a 1. Note that in these formulas it is assumed that all environment setup for both variable reading and setting is done by the receiving processes.

In deriving this formula it is assumed that the setting of 0's and 1's take the same amount of time, and that all transmissions contain an equal distribution of 0's and 1's.

Example 12 – Application of the Bandwidth Estimation Formula

The maximum bandwidths of the two channels of Example 11 can be recalculated by using the above formula, as follows:

The Inode Table Channel

$T_S = [T_S(\text{full} \rightarrow \text{full}) + T_S(\text{full} \rightarrow \text{nonfull}) + T_S(\text{nonfull} \rightarrow \text{nonfull}) + T_S(\text{nonfull} \rightarrow \text{full})]/4$

$= [0 + \text{close}(s) + 0 + \text{open}(s)]/4$

$= (\text{open} + \text{close})/4 = (12 + .2)/4 = 3.05$ (ms)

$T_r = [T_r(\text{full} \rightarrow \text{full}) + T_{env}(\text{full} \rightarrow \text{full}) + T_r(\text{full} \rightarrow \text{nonfull}) + T_{env}(\text{full} \rightarrow \text{nonfull}) + T_r(\text{nonfull} \rightarrow \text{nonfull}) + T_{env}(\text{nonfull} \rightarrow \text{nonfull}) + T_r(\text{nonfull} \rightarrow \text{full}) + T_{env}(\text{nonfull} \rightarrow \text{full})]/4$

$= [\text{open}(f) + 0 + \text{open}(s) + \text{close}(s) + \text{open}(s) + \text{close}(s) + \text{open}(f) + 0]/4$

$= \text{open} + \text{close}/2 = 12.1$ (ms)

Therefore,

$B(0) = 1000/(12.1 + 3.05 + 6) = 47.28$ bits/sec

The Upgraded Directory Channel

$T_S = [T_S(\text{nonempty} \rightarrow \text{nonempty}) + T_S(\text{nonempty} \rightarrow \text{empty}) + T_S(\text{empty} \rightarrow \text{empty}) + T_S(\text{empty} \rightarrow \text{nonempty})]/4$

$= [0 + \text{unlink}(s) + 0 + \text{creat}(s)]/4$

$= (\text{creat} + \text{unlink})/4 = (30 + 22)/4 = 13$ (ms)

$T_r = [T_r(\text{nonempty}\rightarrow\text{nonempty}) + T_{env}(\text{nonempty}\rightarrow\text{nonempty}) + T_r(\text{nonempty}\rightarrow\text{empty})$
$+ T_{env}(\text{nonempty}\rightarrow\text{empty}) + T_r(\text{empty}\rightarrow\text{empty}) + T_{env}(\text{empty}\rightarrow\text{empty})$
$+ T_r(\text{empty}\rightarrow\text{non-empty}) + T_{env}(\text{empty}\rightarrow\text{nonempty})]/4$

$= [\text{rmdir}(f) + 0 + \text{rmdir}(s) + \text{mkdir}(s) + \text{rmdir}(s) + \text{mkdir}(s) + \text{rmdir}(f) + 0]/4$

$= [\text{rmdir}(s) + \text{rmdir}(f)]/2 + \text{mkdir}/2$

$= (180 + 3020)/2 + 260/2 = 1730$ (ms)

Therefore,

$B(0) = 1000/(1730 + 13 + 6) = 0.572$ bits/sec

4.2.3 Differences between the Two Methods

Comparing the results of Examples 11 and 12 one might be tempted to conclude that the two bandwidth computation methods yield similar results for all covert channels. This conclusion, however, is not always the case. Millen's method yields higher bandwidths whenever the times to set up transmission environments and/or those to transmit 0s differ significantly from those to transmit 1s. This may be the case after delays are placed in some but not all TCB primitives of a channel (e.g., in the error return path of a primitive needed to use the channel; this ensures that undue performance penalty is not incurred.). Subsequent recomputation of the (delayed) channel bandwidth by the two methods would yield significantly different results. Experience with using the two methods for Secure Xenix shows that in cases where the times to transmit a 0 and a 1 are close, the two methods yield results that differ by at most 20%.

Millen's method is superior to that presented in [Tsai and Gligor88] not only because it always helps compute the maximum attainable bandwidth but also because during its use one is required to define a realistic scenario of covert channel use. This process helps remove any misunderstandings that might arise when different parties use different assumptions to define the environment set-up times for a channel.

4.3 *TCSEC* REQUIREMENTS AND RECOMMENDATIONS

The *TCSEC* requirements for bandwidth determination of covert channels state, "The system developer . . . shall make a determination (either by actual measurements or by engineering estimation) of the maximum bandwidths of each identified channel."

As explained in Section 4.1, the measurements or estimation of the maximum bandwidth must assume that the covert channels are noiseless, that no processes—other than the sender and receiver—are present in the system when measurements are performed, and that the synchronization time between senders and receivers is negligible. If the channel's bandwidth is estimated using informal methods, measurements of the channel's fastest primitives must be done to determine the values of $T_s(i)$, $T_r(i)$, $T_{env}(i)$ as defined in Section 4.2.2, and the smallest measured value of T_{cs} must be chosen. For both formal and informal bandwidth determination methods, the selection of the TCB primitives measured should be based on realistic scenarios of channel use and should take into account any parameter of TCB state dependency that is relevant for a channel. The system configuration and architecture parameters should be specified for each set of measurements. All measurements necessary for bandwidth determination should be repeatable. Channel aggregation should be considered even though it is not supported by any *TCSEC* requirements or recommendations.

The *TCSEC* requirements for level A1 state, "Formal methods shall be used in the [covert channel] analysis."

In the context of bandwidth measurement or estimation, this requirement suggests that Millen's method (1989a)—defined and illustrated in Section 4.2.1—should be used. Any other relevant information-theory-based method for covert channel bandwidth estimation could be acceptable on a case-by-case basis.

5.0 COVERT CHANNEL HANDLING

In this chapter we present three general methods for the handling of known covert channels that have been proposed and used to date. We also present a set of handling policies based on the analysis of the covert channel threats and risks that is consistent with the objective of the handling guideline of [NCSC TCSEC].

5.1 ELIMINATION OF COVERT CHANNELS

The first method is the *elimination* of covert channels. Elimination requires changing the design and/or implementation of a system to remove covert channels from the system. These changes include:

- The elimination of resource sharing between any potential participants in covert leakage of information by preallocating maximum resource demands to all participants or by partitioning resources on a per-security-level basis;

- The elimination of interfaces, features, and mechanisms which can cause covert leakage.

Example 13 – Elimination of Resource Sharing and Resource Partitioning

To illustrate elimination of covert channels by the elimination of resource sharing, let us reconsider Example 1. The dynamic allocation/deallocation of objects triggers dynamic allocation of memory segments, which provides a resource-exhaustion channel. If the memory is statically partitioned on a per-process or per-security-level basis, the resource-exhaustion channel is eliminated. However, as pointed out in Example 1, this partitioning is not always practical. For example, if the memory (or any other TCB resource, such as internal tables) is partitioned, memory utilization may decrease because some partitions may not be as frequently used as others. This infrequent use may cause a significant degradation in performance. We can find examples of resource partitioning that do not impose undue performance degradation. For instance, the name space of the UNIX® System V interprocess communication objects can be partitioned on a per-security-level basis without significant performance degradation.

Resource partitioning on a per-user, or a per-process, basis is not always possible (e.g., shared hardware resources, such as busses, cannot be partitioned on a per-user or per-process basis). However, the use of these resources can, in principle, be partitioned in time on a per-security-level basis. That is, processes running at the same time can share hardware resources only if the processes run at the same level. For example, in the case of the multiprocessor configurations presented in Example 4, and illustrated in Figure 2-7, a dual-mode process dispatcher can be implemented. In *normal mode*, the use of the CPUs is not partitioned. However, to eliminate the timing channels discussed in Example 4, the processes waiting for service in the "ready" queue(s) can be loaded in available CPUs during the same quantum only if they have the same security level. In this mode, called the *time-partitioned mode*, the timing channels caused by bus or memory locking by each memory reference become harmless. Trusted processes should be exempt from time-partitioned dispatching whenever it can be shown they do not exploit covert channels. (Furthermore, threat analysis performed in the environment of system use may exempt other non-TCB applications from the time-partitioned mode of operation. This exemption is an accreditation-policy matter and, thus, beyond the scope of this guide.)

The performance degradation that may be caused from the time-partitioned dispatching depends on the mix of processes ready to run at any instance. In some environments, where families of processes run concurrently (i.e., are coscheduled, we discuss in Section 2.2.3), the performance degradation will be minimized since all processes of a family run at the same security level. Performance degradation will be significant whenever all processes of the "ready" queue(s) run at different security levels because partitioned-mode dispatching will idle all but one CPU. The overall performance degradation can be mitigated whenever partitioned mode dispatching can be turned on/off selectively by security administrators. In Section 5.4 we discuss policy factors, such as threat analysis, required for such actions.

Example 14 – Elimination of TCB Interfaces, Features, or Mechanisms

In Example 2 we presented a covert channel caused by the UNIX interface convention of preventing the removal of nonempty directories. We argued that eliminating this UNIX convention may be impossible in practice due to user program reliance on the inability to delete nonempty directories. However, in other instances, the elimination of TCB interface conventions, features, or mechanisms causing covert channels is possible. For example, programs can encode classified information by modulating the extent to which they use resources, which is reflected in different accounting (e.g., billing) information returned to users. One could remove this accounting channel by eliminating billing on a user-level basis (i.e., by imposing fixed uniform limits on the extent to which a resource could be used, such as fixed maximum CPU time, fixed maximum I/O time). [Lampson73] Alternatively, this channel can be eliminated by producing accounting information on a per-level basis. Neither alternative seems particularly troublesome, in practice.

5.2 BANDWIDTH LIMITATION

The second method of handling known covert channels is based on *bandwidth-limitation* policies. Such policies require the reduction of the maximum, or alternatively the average, bandwidth of any channel to a predefined acceptable limit. One can limit bandwidths by:

- Deliberately introducing noise into channels (e.g., using random allocation algorithms for shared resources such as indices in shared tables, disk areas, process identifiers; introducing extraneous processes that modify covert channel variables in random patterns); and
- Deliberately introducing delays in each TCB primitive of a real channel.

Example 15 – Introduction of Noise and Delays in Channels

The process identifier channel is an event-count channel that in most systems can have a bandwidth of 10 to 500 bits per second. This channel appears because most TCBs create a new process identifier by incrementing a process-identifier variable whenever a new process is created. Thus, a receiver process could detect whether the sender process transmitted a 0 or a 1 bit by determining whether the identifiers of two processes it creates are consecutive numbers. One can reduce the bandwidth of this channel by changing the process-identifier allocation algorithm of the TCB. That is, the TCB could allocate unused identifiers in the identifier space (pseudo) randomly in a nonmonotonic sequence. Depending upon the randomization characteristics of the allocation algorithm, the bandwidth of the process-identifier channel can be reduced to negligible values. Similar considerations apply to the other allocation algorithms of object identifiers. Note that using random allocation of identifiers introduces negligible overhead and performance degradation in a TCB.

An additional example of noise introduction in covert channels is the notion of "fuzzy time" introduced in [Hu91]. Security kernels can constrain user processes to use only virtual time (i.e., time related only to a user's process activity but not to real time). [Lipner75] To ensure little correlation between real and virtual time by a user process (i.e., a receiver), the relationship between real and virtual time is randomized. This is the underlying principle of the notion of "fuzzy time." The randomization appears to degrade system performance very little (i.e., 5–6% on a VAX system [Hu91]). Thus, "fuzzy time" seems practical even in systems where performance degradation is a significant concern.

An alternative method of reducing channel bandwidths includes the deliberate introduction of spurious processes. That is, user-level processes are introduced in the system to perform random alteration of channel variables. As illustrated in Figure 2-5, processes Up, . . . ,Uq can introduce noise by altering a channel variable. Furthermore, these processes can introduce delays in channels by interposing themselves between the senders and receivers. Analysis presented in [Tsai and Gligor88] shows that the introduction of spurious processes can reduce up to about 75% of the bandwidth of typical channels. However, the introduction of spurious processes for bandwidth-degradation purposes may not be cost-free. Spurious processes tend to degrade system performance—not only channel bandwidth.

The deliberate introduction of delays in TCB primitives of real channels is typically used only for limiting the bandwidth of resource-exhaustion channels. The reason is that one can place delays in these channels in a way that does not degrade system performance until these channels are used. Resource-exhaustion channels make use of resource-exhaustion exception (error) returns to transmit zeros or ones. By placing delays within the return path of an exception, the channel bandwidth is reduced proportionally with the frequency of either the zeros or the ones in the code used by the channel users. In normal mode of TCB operation, however, performance is not degraded because resource-exhaustion exceptions are generally rare (unless channels are used).

It is generally advisable to introduce settable delays within TCB primitives, for two reasons. First, settable delays give system management the opportunity to determine the extent of performance degradation incurred by setting delays selectively on a per-channel basis. Second, whenever the same operating system is used on different hardware platforms, the delay values need to be changed to account for increased or decreased bandwidth for the same channel.

The placement of delays in TCB primitives can be a more complex task than it may first appear. Tradeoffs appear in the placement of delays in the TCB. On one hand, the placement of a delay in functions closer to the TCB interface (i.e., in high-level functions) offers the potential of minimizing the impact of the delay on the entire TCB. For each covert channel, each TCB primitive can be dealt with separately. Thus, one can choose a minimum delay value for each particular TCB primitive and covert channel variable. On the other hand, disadvantages of this delay placement strategy are:

- More coding is needed because for each covert channel, every TCB primitive of the channel would have to be delayed individually; Example 16 illustrates this concept.

- A minimum delay value may not be achievable for every covert channel because, sometimes, it is unclear from the perspective of high-level TCB functions what variables and other functions the low-level functions use. For example, when a user issues a "creat" call in UNIX, the setting of an error message ENFILE, when returning from "copen," may be done either in code using the file structure or in code using the i-nodes. This concept is illustrated in Example 17. In this case, it is impossible to achieve minimum delay for both error situations simultaneously.

Delays could be added in low-level TCB functions common to many TCB primitives. This action is possible because, in practice, each resource within the TCB is managed by a few dedicated functions (resource managers). Thus, all user processes that make use of a resource share these low-level resource-management functions. Delays added to low-level functions will virtually delay all TCB primitives that could take advantage of the corresponding covert channel. However, the disadvantage of this approach is that the length of delay must be determined by the highest-bandwidth channel (e.g., by the fastest TCB primitive) using this low-bandwidth channel. Consequently, TCB primitives used by low-bandwidth channels (or slower TCB primitives that reference the same shared global variable) sometimes tend to be delayed unnecessarily.

Example 16 – Delay Placement in a Resource-Exhaustion Channel

For most covert channels, one must place delays in more than one location of the TCB code. The control-flow paths presented in this example refer to the resource-exhaustion channel provided by the variable "inode→i_flag" of UNIX. This example shows that multiple control paths, both through different TCB primitives of a channel and through the same TCB primitive, must be covered by delays.

Some Control-Flow Paths for the inode_i_flag Variable

creat:

(1) creat→copen→namei→access
(2) creat→copen→access

open:

(1) open→copen→namei→access
(2) open→copen→access

unlink:

(1) unlink
(2) unlink→namei→access

utime:

(1) utime→namei→access
(2) utime→access

chsize:

(1) chsize

This example shows that delay addition to a channel should be placed in low-level TCB functions shared by multiple control paths for the same channel. The low-level TCB functions that are common to all control paths of the inode→ i_flag channel include those of "access," "unlink," and "chsize."

Example 17 – Ambiguity in Delay Placement

The setting of a specific error message within a TCB primitive may correspond to the viewing of multiple covert channels. Thus, the highest-level functions of a TCB primitive cannot determine which channel is being used. Therefore, achieving minimum delay in high-level TCB functions is not always possible. The highest-level functions of TCB primitives where different channels can be distinguished should be found and used for placement of minimum delays. The primitive "creat" of UNIX, in which the ENFILE error is set for both the inode_space and file_table channels, illustrates this case.

File Table Channel

(1) creat→copen→falloc

Inode Space Channel

(1) creat→copen→namei→iget
(2) creat→copen→mknod→ialloc→iget

In event-count channels, the addition of time delays is not advisable. These channels can be used in normal mode of TCB operation rather than in exception cases, and addition of delay would degrade performance significantly. Wherever possible, bandwidth limitation should be achieved by using a randomization algorithm for assigning the next available index or identifier. For TCB primitives that simply read, rather than allocate, indices and identifiers, use of delays may still be necessary whenever the randomization algorithm cannot introduce a sufficient amount of noise to achieve the target bandwidth limit. This situation may arise when the index or identifier range is too small for effective randomization. Example 18 illustrates the selection of randomization points for the process identifier channel of UNIX (variable proc→p_pid of UNIX).

Example 18 – Randomization Points of the Process-Identifier Channel in UNIX

fork

(1) fork

wait

(1) wait
(2) wait→freeproc
(3) wait→sleep→issig→freeproc

getpid

(1) getpid

setpgrp

(1) setpgrp

In the process identifier channel of UNIX, a randomization algorithm should be used within the TCB functions listed below to assign the next available identifier. The invocation of identifier-reading TCB primitives, such as "getpid" and "setpgrp," could also be delayed unconditionally to help limit the channel bandwidth whenever the identifier randomization is inadequate (e.g., provides monotonically increasing identifiers in all cases).

5.3 AUDITING THE USE OF COVERT CHANNELS

The third method of handling *known* covert channels is that of *deterrence* of covert channel use. This method allows all users to exploit known channels but provides a mechanism discouraging channel use. The main deterrence method is channel auditing. This method assumes audit mechanisms can unambiguously detect the use of a channel. Thus, users can be assured of detection of any unauthorized use of covert channels. Note, however, that the *TCSEC* requires only the *ability* to audit covert channels be provided—not that covert channels be actually audited. This detail limits somewhat the effectiveness of audit as a real deterrent.

Covert channel auditing requires that sufficient data be recorded in audit trails to enable the identification of (1) individual covert channel use, or use of certain channel types; and (2) identification of the senders and receivers of individual channels or of channel types (i.e., the identification of the covert channel users). Furthermore, discovery of covert channel use must be certain (i.e., covert channel auditing must not be circumventable), and false detection of covert channel use must be avoided. Circumvention of covert channel auditing is undesirable because it allows leakage of information to remain undetected. False detection of covert channel use is also undesirable because it may make it impossible to distinguish between innocuous user activity and covert channel use.

Estimation of actual covert channel bandwidth is possible and desirable once covert channel use has been determined by audit-trail analysis. Note that, in general, it is impossible to discover the actual information being leaked through covert channels from audit trails because a user can encrypt it before leakage. Also, one cannot distinguish between real information and noise leakage merely by inspecting audit trails. Constant streams of either zeros or ones are the only recorded patterns one can unambiguously classify as noise.

Most of the problems identified in covert channel auditing are fundamental and are shared by most operating systems; these problems include (1) inability of distinguishing use of covert channels from innocuous use of TCB primitives, and (2) ambiguity in distinguishing senders from receivers among covert channel users. These problems appear because single TCB primitives may both alter and view a variable or attribute, depending on the argument values of that primitive and on the system state, and because different TCB primitives may be shared by different covert channels. Such primitives allow users to disguise covert channel use, thereby circumventing audit, and cause false detection of covert channels [Shieh and Gligor90]. Figures 3-2 and 3-4 show examples of such primitives.

Key design concerns of covert channel auditing are those of determining what events should be recorded by auditing mechanisms and what data should be maintained by auditing tools to ensure that all covert channel use can be discovered. The identification of covert channels can be summarized as sets of <variable, PA_h,PV_i> triples (where PA_h/PV_i represents a TCB primitive altering/viewing the variable, as shown in Figures 3-2 and 3-4), suggesting that recording all events including pairs of <PA_h, variable> and <PV_i, variable> is necessary and sufficient for covert channel auditing. However, recording such events is fraught with both practical and fundamental difficulties because audit-record formats and mechanisms currently used in practice include only process identifiers, object identifiers, process and object security levels, type of event (e.g., primitive identifier), and event outcome (i.e., success or error value); viz., Refs. [NCSC TCSEC, NCSC Audit]. Fields for recording covert channel variables are not included in existent audit-record formats. [Shieh and Gligor90] provide examples of such fields and their setting.

In resource-exhaustion channels one can sometimes identify $<PV_i$, variable$>$ pairs from recorded primitive identifiers and event outcomes. For example, whenever the event outcome is an error that can be unambiguously associated with a channel variable, the auditor can infer that the recorded primitive identifier represents PV_i. However, whenever the event outcome is no-error and if $PV_i = PA_h$, the auditor cannot tell whether the recorded primitive identifier is for a PV_i- or a PA_h-type of primitive; nor can the auditor tell whether the recorded primitive identifier represents a PA_h type of primitive or an innocuous TCB primitive, whenever PV_i != PA_h occurs. Whether the use of a TCB primitive is innocuous or covert channel related depends on the state of the system and on the values of the primitive's parameters. Thus, the recording of channel variables is necessary for all no-error outcomes of a primitive associated with a covert channel.

Fundamental difficulties with recording channel variables appear because many TCB primitives are shared by several covert channels. Thus, a PA_h- or a PV_i-type primitive may refer to variables of multiple channels. The actual use of a single variable cannot be discerned even when all potential variables are known and even when the error outcomes of a PV_i primitive can be unambiguously associated with single channel variables. For example, a user can infer a no-error outcome of a shared PV_i primitive on a given variable from an error outcome of the same PV_i primitive on another variable. This process enables users to disguise the use of a channel as transmission of noise (e.g., constant strings of 0s or 1s) on multiple channels (an example of reference [Shieh and Gligor90]). In such cases, auditors have to maintain additional information to enable the detection of all potential use of covert channels.

Examples of the storage channel auditing problems mentioned above have been illustrated in the context of the Secure Xenix system in reference [Shieh and Gligor90]. Solutions to these problems are also presented in that reference. We must note, however, that not all use of covert channels can be audited. Example 4 of section 2.2 illustrates a few instances of covert timing channels usage where auditing is impractical.

5.4 *TCSEC* REQUIREMENTS AND RECOMMENDATIONS

TCSEC requirements for covert channel handling are included in the audit and documentation requirements. Section 8 of the *TCSEC*, "A Guideline on Covert Channels," makes additional recommendations.

The audit requirements of the *TCSEC* state, "The TCB shall be able to audit the identified events that may be used in the exploitation of covert storage channels."

The design documentation requirements state:

> [Documentation] shall also present the results of the covert channel analysis and the trade-offs involved in restricting the channels. All auditable events that may be used in the exploitation of known covert storage channels shall be identified. The bandwidths of known covert channels, the use of which is not detectable by the auditing mechanisms, shall be provided.

The [NCSC TCSEC] guidelines on covert channels suggest the following combination of the above methods: (1) use elimination methods wherever possible to eliminate channels with bandwidths over 0.1 bits/second; (2) use bandwidth-limitation methods to reduce, whenever possible, the maximum bandwidth of every channel that cannot be eliminated to 1 bit/second or less; (3) use deterrence methods, namely audit, for channels with bandwidths over 0.1 bit/second; and (4) use a "don't care" policy for covert channels with bandwidths less than 0.1 bit/second.

The *TCSEC* requirements for handling covert channels and the covert channel guidelines presented in Section 8 of the *TCSEC* suggest the following handling policy:

- Covert channels with bandwidths under some predefined lower limit b are acceptable;
- Covert storage channels with bandwidths over lower limit b shall be auditable; the bandwidths of all storage channels that are not auditable shall be documented;
- Covert channels with bandwidths over some predefined upper limit B > b represent a significant threat and, wherever possible, they should either be eliminated or their bandwidth should be reduced to B bits/second; and

- Covert storage channels with bandwidths over b bits per second should be audited; this gives system administrators the ability to detect and procedurally correct significant compromise.

This policy allows for the existence of *storage channels* that are *not* auditable. Also, it allows for the possibility that covert storage and timing channels with bandwidths over B = 1 bit/second will exist in secure systems. However, the suggested values of b = 0.1 bits/second and B = 1 bit/second are not justified based on any specific policy. The only basis for deriving these values is the determination that:

- Covert channel handling may impose performance penalties, and that bandwidths of 1 bit/second are acceptable in most environments; and

- Although covert channels with bandwidth of over 1 bit/second may be allowed in a secure system, covert channels with bandwidths of over 100 bits/second approximate the rate at which many (old) computer terminals are run (or users can type). Therefore, the existence of such channels in a secure computer system would seem inappropriate.

(Note: This guide may not contain the current covert channel bandwidth policy, which is subject to change. Please contact the NCSC for information about the current policy.)

5.5 HANDLING POLICIES BASED ON THREAT ANALYSIS

Although the intent of the *TCSEC* handling requirements and guideline is sound, the justification of the particular values of bandwidth limits b (0.1 bits/second) and B (1 bit/second) may be less than satisfactory for the following reasons:

- The threat posed by covert channels depends on the specific application environment of use; therefore, the appropriateness of the bandwidth limits b and B cannot be determined without threat analyses within the specific application environment. Hence, these limits cannot be specified during the design or evaluation process. Whenever practical, a system should include variable covert channel delays whose values can be set by security administrators [IBM87].

- The threat posed by covert channels depends on the characteristics of the covert channels themselves. For example, (1) some covert channels have a maximum attainable value that may be very high but the noise and delay under normal system load decrease the attainable maximum bandwidth of these channels under an acceptable limit B; (2) some covert channels can be exploited more readily than others having simpler scenarios of use; (3) some covert channels cannot be audited because they appear at low system levels where audit is impractical; or (4) some covert channels can be aggregated serially or in parallel, increasing the effective bandwidth available to senders and receivers, and some others cannot be aggregated [Tsai and Gligor88].

These considerations indicate that the values of the bandwidth limits b and B can only be determined after a threat analysis which includes the above factors. Let us consider an example illustrating the necessity of threat analysis in the environment of secure system use.

Example 19 – Application Dependency of Bandwidth Limits

Consider an application environment in which classified satellite images are processed (e.g., satellite images of various agricultural crops in certain countries). Each image frame consists of 512x512 picture elements (pixels), each pixel having 8 bits, and each application includes up to 10,000 frames. A multilevel secure system is used which includes a covert channel of 10,000 bits/second. This means an image frame can be declassified by using this covert channel in approximatively 200 seconds. Thus, up to 18 frames can be declassified in an unauthorized manner per hour. The need to operate this channel for more than one hour to declassify less than 0.2% of the data makes this threat negligible. The likelihood of detecting the use of this channel by (off-line) audit is very high due to its long period of operation. Thus, in this environment b can be set to 10,000 bits/second or even higher. Of course, information concerning the source of the satellite images may have a higher classification. The vulnerability of this information to covert channels may require its separate processing by trusted software rather than by untrusted application code.

In contrast, consider an application environment where 64-bit encryption keys are generated whose lifetimes are comparable with that of a login session (i.e., 8 hours). Even if these keys are encrypted when stored on nonvolatile storage, their actual use by application software would be in cleartext form. If the secure system used in this application contains a 0.1 bit/second channel, each session key can be declassified in less than 11 minutes, rendering the key vulnerable for most of its lifetime. The likelihood of being able to detect the use of this channel through off-line audit may not be very high because of the relatively short period of channel use. On-line audit of this channel may be even less likely. Thus, in this application environment B could not be set to 0.1 bit/second. Instead, a B of 0.002 bit/second would seem more appropriate because, at that rate, it would take at least 8 hours to declassify a key.

The classification range of the information processed in a trusted system and, therefore, the trusted system class (i.e., B2–A1) must also be considered in threat analysis. Covert channels of high bandwidths (e.g., 1,000–10,000 bits per second) may be acceptable in a B2 system in which only Top Secret and Secret information is processed, and leakage below the Secret level is impossible. In contrast, the same leakage rate may be unacceptable in A1 systems that process multilevel information, since the possibility of unauthorized declassification of Top Secret information might be a real threat.

In threat analyses, one must also consider the characteristics of each covert channel. For example, the CPU scheduling channels of Example 3 may have a maximum bandwidth of 5–300 bits/second on systems comparable to today's fast workstations (depending on the operating system and scheduling parameters [Huskamp78]). However, compared with the upgraded directory channels, the CPU scheduling channels are much more difficult to use in any real system due to lack of control over scheduling parameters and due to noise introduced by background processes. Thus, these channels (and also those illustrated in Example 4 which use shared hardware resources) are significantly less likely to be used in practice than the noiseless upgraded-directory channels of Example 2. On the other hand, other noisy channels such as the various identifier channels may be more likely to be used than the upgraded directory channels because the likelihood of auditing correctly a noiseless channel is higher than that of auditing correctly a noisy channel. Thus, the high likelihood of detecting the use of the upgraded directory channel may deter its use.

This example indicates the need for establishing a threat-analysis policy on a per environment and system basis. It also suggests this analysis cannot be carried out at system evaluation time without postulating the characteristics of the application environment. Finally, this example suggests few of the important parameters that should be considered for such an analysis.

6.0 COVERT CHANNEL TESTING

6.1 TESTING REQUIREMENTS AND RECOMMENDATIONS

The *TCSEC* requirements of test documentation at class B2 state, ". . . [Test documentation] shall include results of testing the effectiveness of the methods used to reduce covert channel bandwidths."

Covert channel testing demonstrates that covert channel handling methods chosen by system designers work as intended. These methods include covert channel elimination, bandwidth limitation, and (ability to) audit. Testing is also useful to confirm that potential covert channels discovered in the system are in fact real channels. Furthermore, testing is useful when the handling method for covert channels uses variable bandwidth-reduction parameters (e.g., delays) that are settable by system administrators (e.g., by auditors).

Bandwidth estimation methods necessary for the handling of covert channels may be based on engineering estimation rather than on actual measurements. Bandwidth estimations provide upper bounds for covert channels before employing any handling methods. In contrast, covert channel testing always requires doing actual measurements to determine the covert channel bandwidths after implementing the chosen handling method in a system.

6.2 TEST DOCUMENTATION

Test plan documentation, including test conditions, test environment set-up, test data, expected test outcome, and actual test result documentation are discussed in the security testing guideline [NCSC Testing] in detail. Therefore, we do not repeat the discussion here. The security testing guideline also gives an example of the test plans for a real channel (i.e., for the upgraded-directory channel of Example 2).

7.0 SATISFYING THE *TCSEC* REQUIREMENTS FOR COVERT CHANNEL ANALYSIS

We present in this chapter the *TCSEC* requirements relevant to covert channel analysis and suggest ways to satisfy them. For each class containing them, we show the requirements of CCA (which include channel identification, bandwidth measurement or estimation, audit, and design documentation). We also summarize the recommendations made in the *TCSEC* guidelines on covert channels. Our recommendations, though derived from *TCSEC* objectives, are not requirements.

7.1 REQUIREMENTS FOR CLASS B2

7.1.1 Covert Channel Analysis

Channel Identification

The *TCSEC* requirement for CCA states, "The system developer shall conduct a thorough search for covert storage channels . . ."

Developers shall identify the sources of information used to satisfy this requirement. These sources shall include system reference manuals and the DTLS. They should include source code and processor specifications whenever the identification method includes source code and hardware analysis. Developers should show the identification method they use to be sound and reliable (e.g., repeatable). This implies, among other things, that independent evaluators can use the method on the same sources of covert channel information and get the same results. Otherwise, the identification evidence will lack credibility.

Bandwidth Measurement or Engineering Estimation

The *TCSEC* requirement for this area states, "The system developer shall . . . make a determination (either by actual measurement or by engineering estimation) of the maximum bandwidth of each identified channel."

In measuring or estimating covert channel bandwidth, developers should consider the following factors (as discussed in Section 4.1):

- For maximum bandwidth, assume the channel is noiseless and the presence of other processes in the system do not delay the senders and receivers.

- The choice of informal estimation methods requires defining (and possibly justifying) assumptions about the coding method and, therefore, the distribution of 0s and 1s in all transmissions. Whenever possible, use Millen's information-theory-based method, which yields the maximum bandwidth and also provides the required coding method to achieve it [Millen89a].

- Covert channel measurements should include the fastest primitives for altering, viewing, and setting up the transmission environment. Also, bandwidth measurements should involve the demonstrably fastest process (context) switch time.

- To determine bandwidth, derive the TCB primitives to measure from real scenarios of covert channel use. Take into account parameter and TCB state dependencies of each selected primitive (if any).

- Specify the measurement environment. This specification includes (1) the speed of the system components, (2) the system configuration, (3) the sizes of the memory and cache components, and (4) the system initialization. Document the sensitivity of the measurement results to configuration changes. (This documentation enables accreditors to assess the real impact of covert channels in different environments of use.)

- Sender-receiver synchronization time may be considered negligible and, therefore, ignored.

- Consider channel aggregation in bandwidth estimation.

- All measurements must be repeatable.

7.1.2 Audit

The *TCSEC* Audit requirements state, ". . . The TCB shall be able to audit the identified events that may be used in the exploitation of covert storage channels."

To satisfy this requirement, audit mechanisms should include the following features whenever possible (viz., discussion of Section 5.3):

- The audit record should include the storage channel variables.

- The audit code of the TCB should cover all control paths leading to the alteration or viewing of the storage channel variables.
- The audit record should include sufficient information to identify unambiguously the senders and receivers of a storage channel.
- The audit mechanism should be noncircumventable whenever audit is turned on for a covert channel.

7.1.3 Design Documentation

Part of the *TCSEC* requirements for this area states the following:

> . . . This documentation shall also present the results of the covert channel analysis and the tradeoffs involved in restricting the channels. All auditable events that may be used in the exploitation of known covert storage channels shall be identified. The bandwidths of known covert storage channels, the use of which is not detectable by the auditing mechanism, shall be provided. . . .

Documentation of Identified Channels

The documentation of each identified storage channel should consist of the variable the channel views/alters and the TCB primitives that alter or view the variable. Developers should distinguish potential covert channels from real ones.

Documentation of Bandwidth Estimation

Developers must document measurements of each covert channel primitive and should include the bandwidth computation for each channel. They should document the measurement environment as specified in Section 7.1.1.

Documentation of Covert Channel Auditing

Documentation shall include a definition of each event used in the exploitation of a covert channel. This documentation should cite the definition of the TCB primitives and TCB paths leading to a covert channel variable. Developers should also identify and justify the covert storage channels that cannot be audited.

Channel Restriction and Handling Policies

The documentation shall include a description of (1) how covert channels are eliminated, and (2) how covert channel bandwidth is limited to a value deemed acceptable. Sections 5.1 and 5.2 provide a discussion of channel restriction methods.

Covert channel-handling policies should be consistent with the intent of the *TCSEC* guidelines. Covert channel bandwidth limits (b, B)—as defined in Sections 5.4 and 5.5—are considered outside the purview of the *TCSEC*. System accreditors should specify these limits for the specific threat environment the system will be used in.

7.1.4 Test Documentation

Part of the *TCSEC* requirements for test documentation states, ". . . [Test documentation] shall include results of testing the effectiveness of the methods used to reduce covert channel bandwidths."

See the security testing guideline [NCSC Testing], which discusses these requirements.

7.2 ADDITIONAL REQUIREMENTS FOR CLASS B3

Class B3 incorporates all requirements of class B2. In addition, the following requirements apply.

7.2.1 Covert Channel Analysis

Channel Identification

The only additional B3-class requirement is the identification of timing channels. Developers must define timing channel scenarios and identify all system components providing independent sources of timing (e.g., CPUs and I/O processors). Developers may use the same sources of information and methods for identifying timing channels as those used for identifying storage channels.

Bandwidth Measurement or Engineering Estimation

There are no additional requirements.

7.2.2 Audit

There are no additional requirements.

7.2.3 Design Documentation

Documentation of Identified Channels

The only additional requirement for this class is the documentation of all timing channels. Developers should document these channels by specifying the variable of the TCB state that may be changed by direct or indirect actions of user processes. These channels include CPU-scheduling channels, I/O-processor-scheduling channels, and page-replacement channels.

7.2.4 Test Documentation

There are no additional requirements.

7.3 ADDITIONAL REQUIREMENTS FOR CLASS A1

Class A1 contains all the class B3 requirements. The only additional requirements of class A1 appear in CCA.

Covert Channel Analysis

Part of the *TCSEC* requirement for this area states, ". . . Formal methods shall be used in analysis."

Channel Identification

Developers can apply formal methods on both formal specifications and source code of the TCB. These methods include syntactic information-flow analysis (with or without the use of semantic analysis) and noninterference analysis. Developers shall apply the chosen identification method to the FTLS. Unless the identification of covert channels is made a part of the specification-to-code correspondence (in which case source-code analysis is included), developers should complement the FTLS analysis with formal or informal source-code analysis. Otherwise, covert channels may remain undetected.

Bandwidth Measurement or Engineering Estimation

The requirement to use formal methods suggests that developers should use Millen's method (1989a)—defined and illustrated in Section 4.2.1. Any other information-theory-based method for covert channel bandwidth estimation may be acceptable on a case-by-case basis.

ACRONYMS AND ABBREVIATIONS

AIS	Automated Infomation System
CCA	Covert Channel Analysis
CPU	Central Processing Unit
DAC	Discretionary Access Control
DoD	Department of Defense
DTLS	Descriptive Top-level Specification
EHDM	Enhance Hierarchical Development Methodology
FDM	Formal Development Methodology
FTLS	Formal Top-level Specification
GVE	Gypsy Verification Environment
HDM	Hierarchical Development Methodology
I/O	Input/Output
IPC	InterProcess Communication
LOCK	Logical Co-processing Kernel
LRU	Least Recently Used
MAC	Mandatory Access Control
MLS	Multilevel Secure
NCSC	National Computer Security Center
SAT	Secure Ada Target
SRM	Shared Resource Matrix

TCB	Trusted Computing Base
TCSEC	Trusted Computer System Evaluation Criteria
TLS	Top-level Specification

GLOSSARY

ACCESS

Ability and means to communicate with (i.e., input to or receive output from) or otherwise make use of any information, resource, or component in an AIS. NOTE: An individual does not have "access" if the proper authority or a physical, technical, or procedural measure prevents them from obtaining knowledge or having an opportunity to alter information, material, resources, or components.

ACCESS TYPE

Privilege to perform an action on a program or file. NOTE: Read, write, execute, append, modify, delete, and create are examples of access types.

ACCREDITATION

Formal declaration by a designated approving authority that an AIS is approved to operate in a particular security mode using a prescribed set of safeguards.

ADMINISTRATIVE USER

A user assigned to supervise all or a portion of an AIS.

AUDIT

Independent review and examination of records and activities to assess the adequacy of system controls, to ensure compliance with established policies and operational procedures, and to recommend necessary changes in controls, policies, or procedures.

AUDIT MECHANISM

The processes used to collect, review, and/or examine system activities.

AUDITOR

An authorized individual, or role, with administrative duties, which include selecting the events to be audited on the system, setting up the audit parameters which enable the recording of those events, and analyzing the trail of audit events.

AUDIT TRAIL

Chronological record of system activities to enable the reconstruction and examination of the sequence of events and/or changes in an event.

BANDWIDTH

A characteristic of a communication channel that is the amount of information that can be passed through it in a given amount of time, usually expressed in bits per second.

BELL-LA PADULA SECURITY MODEL

Formal-state transition model of a computer security policy that describes a formal set of access controls based on information sensitivity and subject authorizations.

CATEGORY

Restrictive label that has been applied to both classified and unclassified data, thereby increasing the requirement for protection of, and restricting the access to, the data. NOTE: Examples include sensitive compartmented information, proprietary information, and North Atlantic Treaty Organization information. Individuals are granted access to special category information only after being granted formal access authorization.

CERTIFICATION

Comprehensive evaluation of the technical and nontechnical security features of an AIS and other safeguards, made in support of the accreditation process, to establish the extent to which a particular design and implementation meets a set of specified security requirements.

CHANNEL

An information transfer path within a system. May also refer to the mechanism by which the path is effected.

COVERT CHANNEL

Unintended and/or unauthorized communications path that can be used to transfer information in a manner that violates an AIS security policy. See also: Covert Storage Channel, Covert Timing Channel.

COVERT STORAGE CHANNEL

Covert channel that involves the direct or indirect writing to a storage location by one process and the direct or indirect reading of the storage location by another process. NOTE: Covert storage channels typically involve a finite resource (e.g., sectors on a disk) that is shared by two subjects at different security levels.

COVERT TIMING CHANNEL

Covert channel in which one process signals information to another process by modulating its own use of system resources (e.g., central processing unit time) in such a way that this manipulation affects the real response time observed by the second process.

DATA

Information with a specific physical representation.

DATA INTEGRITY

Condition that exists when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.

DESCRIPTIVE TOP-LEVEL SPECIFICATION (DTLS)

Top-level specification that is written in a natural language (e.g., English), an informal program design notation, or a combination of the two. NOTE: Descriptive top-level specification, required for a class B2 and B3 AIS, completely and accurately describes a trusted computing base. See formal top-level specification.

DISCRETIONARY ACCESS CONTROL

Means of restricting access to objects based on the identity and need-to-know of users and/or groups to which the object belongs. NOTE: Controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (directly or indirectly) to any other subject. See mandatory access control.

DOMAIN

Unique context (e.g., access control parameters) in which a program is operating; in effect, the set of objects that a subject has the ability to access.

EXPLOITABLE CHANNEL

Covert channel that is intended to violate the security policy governing an AIS and is usable or detectable by subjects external to the trusted computing base. See covert channel.

FORMAL SECURITY POLICY MODEL

Mathematically precise statement of a security policy. NOTE: Such a model must define a secure state, an initial state, and how the model represents changes in state. The model must be shown to be secure by proving that the initial state is secure and that all possible subsequent states remain secure.

FORMAL TOP-LEVEL SPECIFICATION (FTLS)

Top-level specification that is written in a formal mathematical language to allow theorems, showing the correspondence of the system specification to its formal requirements, to be hypothesized and formally proven. NOTE: Formal top-level specification, required for a class A1 AIS, completely and accurately describes the trusted computing base. See descriptive top-level specification.

FUNCTIONAL TESTING

Segment of security testing in which advertised security mechanisms of an AIS are tested under operational conditions.

MANDATORY ACCESS CONTROL

Means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. See discretionary access control.

MULTILEVEL DEVICE

Device that is trusted to properly maintain and separate data of different security levels.

OBJECT

Passive entity that contains or receives information. NOTE: Access to an object implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, and network nodes.

OVERT CHANNEL

Communications path within a computer system or network that is designed for the authorized transfer of data. See covert channel.

PROCESS

A program in execution. See domain and subject.

READ

Fundamental operation in an AIS that results only in the flow of information from an object to a subject. See access type.

READ ACCESS

Permission to read information in an AIS.

SECURITY ADMINISTRATOR

An administrative role (or user) responsible for the security of an AIS and having the authority to enforce the security safeguards on all others who have access to the AIS (with the possible exception of the auditor.) Also called system administrator.

SECURITY LEVEL

The combination of a hierarchical classification and a set of nonhierarchical categories that represents the sensitivity of information.

SECURITY POLICY

The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

SECURITY POLICY MODEL

An informal or formal presentation of a security policy enforced by the AIS. It must identify the set of rules and practices that regulate how an AIS manages, protects, and distributes sensitive information. See Bell-La Padula security model and formal security policy model.

SECURITY TESTING

Process to determine that an AIS protects data and maintains functionality as intended. NOTE: Security testing may reveal vulnerabilities beyond the scope of the AIS design.

SUBJECT

Active entity in an AIS, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. NOTE: Technically, a process/domain pair.

SUBJECT SECURITY LEVEL

Sensitivity label(s) of the objects to which the subject has both read and write access. NOTE: Security level of a subject must always be dominated by the clearance level of the user with which the subject is associated.

TCB PRIMITIVE

An operation implemented by the TCB whose interface specifications (e.g., names, parameters, effects, exceptions, access control checks, errors, and calling conventions) are provided by system reference manuals or DTLS/FTLS as required.

TOP-LEVEL SPECIFICATION (TLS)

A nonprocedural description of system behavior at the most abstract level; typically, a functional specification that omits all implementation details.

TROJAN HORSE

Computer program containing an apparent or actual useful function that contains additional (hidden) functions that allows unauthorized collection, falsification or destruction of data.

TRUSTED COMPUTING BASE (TCB)

Totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy. NOTE: The ability of a trusted computing base to enforce correctly a unified security policy depends on the correctness of the mechanisms within the trusted computing base, the protection of those mechanisms to ensure their correctness, and the correct input of parameters related to the security policy.

USER

Person or process accessing an AIS by direct connections (e.g., via terminals) or indirect connections. NOTE: "Indirect connection" relates to persons who prepare input data or receive output that is not reviewed for content or classification by a responsible individual.

VERIFICATION

The process of comparing two levels of an AIS specification for proper correspondence (e.g., security policy model with top-level specification, top-level specification with source code, or source code with object code). NOTE: This process may or may not be automated.

WRITE

Fundamental operation in an AIS that results only in the flow of information from a subject to an object. See access type.

WRITE ACCESS

Permission to write to an object in an AIS.

REFERENCES

[Andrews and Reitman80]

G. R. Andrews and R. P. Reitman, "**An Axiomatic Approach to Information Flow in Programs**," *ACM Transactions on Programming Languages and Systems*, 2:1, pp. 56–76, January 1980.

[Bach86]

M. J. Bach, ***The Design of the UNIX Operating System***, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

[Bell and La Padula76]

D. E. Bell and L. J. La Padula, ***Secure Computer System: Unified Exposition and Multics Interpretation***, The MITRE Corp., Report No. MTR-2997 Revision 1, Electronic Systems Division, U. S. Air Force Systems Command, Technical Report ESD-TR-75-306, Bedford, Massachusetts, March 1976 (available as NTIS AD-A023588).

[Benzel84]

T. V. Benzel, "**Analysis of a Kernel Verification**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 125–131, April 1984.

[Biba77]

K. J. Biba, ***Integrity Considerations for Secure Computer Systems***, The MITRE Corp., Report No. MTR-3153 Revision 1, Electronic Systems Division, U. S. Air Force Systems Command, Technical Report ESD-TR-76-372, Bedford, Massachusetts, April 1977.

[Boebert85]

W. E. Boebert, R. Y. Kain, and W. D. Young, "**Secure Computing: The Secure Ada Target Approach**," *Scientific Honeyweller*, 6:2, pp. 1–17, July 1985.

[Cipher90]

IEEE Computer Society Technical Committee on Security and Privacy, ***Minutes of the First Workshop on Covert Channel Analysis***, *Cipher* Newsletter, Special Issue, pp. 8–12, July 1990.

[Clark and Wilson87]

D. D. Clark and D. R. Wilson, "**A Comparison of Commercial and Military Computer Security Policies**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 184–194, April 1987.

[Denning76]

D. E. Denning, "**A Lattice Model of Secure Information Flow**," *Communications of the ACM*, 19:5, pp. 236–243, May 1976.

[Denning77]

D. E. Denning and P. J. Denning, "**Certification of Programs for Secure Information Flow**," *Communications of the ACM*, 20:7, pp. 504–513, July 1977.

[Denning83]

D. E. Denning, ***Cryptography and Data Security***, Addison-Wesley, Reading, Massachusetts, 1983 (reprinted).

[DoD Directive]

Department of Defense, ***Security Requirements for Automated Information Systems (AISs)***, DoD Directive 5200.28, 21 March 1988.

[Eckmann87]

S. T. Eckmann, "**Ina Flo: The FDM Flow Tool**," *Proceedings of the 10th National Computer Security Conference*, Baltimore, Maryland, pp. 175–182, September 1987.

[Feiertag80]

R. Feiertag, ***A Technique for Proving Specifications are Multilevel Secure***, Technical Report CSL-109, Computer Science Laboratory, SRI International, Menlo Park, California, January 1980.

[Gallager68]

R. G. Gallager, ***Information Theory and Reliable Communications***, John Wiley and Sons, New York, 1968.

[Gasser88]

M. Gasser, ***Building A Secure Computer System***, Van Nostrand Reinhold, New York, 1988.

[Gligor86]

V. D. Gligor and C. S. Chandersekaran, "**Towards the Development of Secure Distributed Systems**," in Grissonnanche, A. (editor), *Information Security: The Challenge*, IFIP Press, Monte Carlo, pp. 395–406, 1986.

[Gligor87]

V. D. Gligor, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W.-D. Jiang, A. Johri, G. L. Luckenbaugh, N. Vasudevan, "**Design and Implementation of Secure Xenix**," *IEEE Transactions on Software Engineering*, 13:2, pp. 208–221, February 1987.

[Goguen and Meseguer82]

J. A. Goguen and J. Meseguer, "**Security Policies and Security Models**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 11–20, April 1982.

[Goguen and Meseguer84]

J. A. Goguen and J. Meseguer, "**Unwinding and Inference Control**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 75–86, April 1984.

[Haberman72]

A. N. Haberman, "**Synchronization of Communicating Processes**," *Communications of the ACM*, 12:7, pp. 171–176, July 1972.

[Haigh87]

J. T. Haigh, R. A. Kemmerer, J. McHugh, and W. D. Young, "**An Experience Using Two Covert Channel Analysis Techniques on a Real System Design**," *IEEE Transactions on Software Engineering*, 13:2, pp. 157–168, February 1987.

[Haykin83]

S. Haykin, ***Communication Systems***, John Wiley and Sons, New York, 1983 (second edition).

[He and Gligor90]

J. He and V. D. Gligor, "**Information Flow Analysis for Covert-Channel Identification in Multilevel Secure Operating Systems**," *Proceedings of the 3rd*

IEEE Workshop on Computer Security Foundations, Franconia, New Hampshire, pp. 139–148, June 1990.

[Honeywell85a]

Honeywell Information Systems Inc., ***SCOMP Interpretation of the Bell-La Padula Model***, Technical Report, McLean, Virginia, March 1985.

[Honeywell85b]

Honeywell Information Systems Inc., ***Multics Security Model - Bell and La Padula***, Multics Design Document MDD-002, Cambridge, Massachusetts, August 1985.

[Hu91]

W.-M. Hu, "**Reducing Timing Channels with Fuzzy Time,**" *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, pp. 8–20, May 1991.

[Huskamp78]

J. C. Huskamp, ***Covert Communication Channels in Timesharing Systems***, Technical Report UCB-CS-78-02, Ph.D. Thesis, University of California, Berkeley, California, (1978).

[IBM87]

IBM Corp., ***Secure Xenix, version 1.1 - Descriptive Top-Level Specifications***, June 1987.

[Jones79]

A. K. Jones, R. K. Chansler Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "**StarOS: A Multiprocessor Operating System for the Support Task Forces,**" *Proceedings of the 7th Symposium on Operating System Principles*, Pacific Grove, California, pp. 117–127, December 1979.

[Karger87]

P. A. Karger, "**Limiting the Damage Potential of Discretionary Trojan Horses,**" *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 32–37, April 1987.

[Karger and Wray91]

P. A. Karger and J. C. Wray, "**Storage Channels in Disk Arm Optimization**," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, pp. 52–61, May 1991.

[Kemmerer83]

R. A. Kemmerer, "**Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels**," *ACM Transactions on Computer Systems*, 1:3, pp. 256–277, August 1983.

[Kemmerer86]

R. A. Kemmerer, ***Verification Assessment Study Final Report***, National Computer Security Center, Technical Report C3-CR01-86, Library No. S-228,204, March 1986.

[Lampson73]

B. W. Lampson, "**A Note on the Confinement Problem**," *Communications of the ACM*, 16:10, pp. 613–615, October 1973.

[Leach83]

P. J. Leach, P. H. Levine, B. P. Dorous, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, "**The Architecture of an Integrated Local Network**," *IEEE Journal on Selected Areas in Communications*, 1:5, pp. 842–856, November 1983.

[Lipner75]

S. B. Lipner, "**A Comment on the Confinement Problem**," *Operating Systems Review*, 9:5, pp. 192–196, November 1975.

[Loepere85]

K. Loepere, "**Resolving Covert Channels within a B2 Class Secure System**," *Operating Systems Review*, ACM SIGOPS, 19:3, pp. 9–28, July 1985.

[Luckenbaugh86]

G. L. Luckenbaugh, V. D. Gligor, L. J. Dotterer, C. S. Chandersekaran, and N. Vasudevan, "**Interpretation of the Bell-La Padula Model in Secure Xenix**," *Proceedings of the 9th National Computer Security Conference*, Gaithersburg, Maryland, pp. 113–125, September 1986.

[McHugh and Akers87]

J. McHugh and R. L. Akers, ***A Formal Justification for the Gypsy Information Flow Tool***, Technical Report, Computational Logic Inc., Austin, Texas, 1987.

[McHugh and Good85]

J. McHugh and D. I. Good, "**An Information Flow Tool for Gypsy**," *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 46–48, April 1985.

[Millen76]

J. K. Millen, "**Security Kernel Validation in Practice**," *Communications of the ACM*, 19:5, May 1976.

[Millen78]

J. K. Millen, "**An Example of a Formal Flow Violation**," *Proceedings of the IEEE International Conference on Computer Software and Applications*, Chicago, Illinois, pp. 204–208, 1978.

[Millen81]

J. K. Millen, "**Information Flow Analysis of Formal Specifications**," *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 3–8, April 1981.

[Millen89a]

J. K. Millen, "**Finite-State Noiseless Covert Channels**," *Proceedings of the Computer Security Foundations Workshop*, Franconia, New Hampshire, pp. 81–85, June 1989.

[Millen89b]

J. K. Millen, ***Foundations of Covert-Channel Detection***, The MITRE Corporation, Technical Report MTR-10538, January 1989.

[NCSC Audit]

National Computer Security Center, ***A Guide to Understanding Audit in Trusted Systems***, NCSC-TG-001, version 2, June 1988.

[NCSC DAC]

National Computer Security Center, ***A Guide to Understanding Discretionary Access Control***, NCSC-TG-003 , version 1, 30 September 1987.

[NCSC TCSEC]

National Computer Security Center, ***Department of Defense Trusted Computer System Evaluation Criteria***, DoD 5200.28-STD, December 1985.

[NCSC Testing]

National Computer Security Center, ***A Guide to Understanding Security Testing and Test Documentation in Trusted Systems***, Draft, 1989.

[Osterhout80]

J. K. Osterhout, S. H. Scelza, and P. S. Sinhu, "**Medusa: An Experiment in Distributed Operating System Structure**," *Communications of the ACM*, 23:2, pp. 92–105, February 1980.

[Osterhout82]

J. K. Osterhout, "**Scheduling Techniques for Concurrent System Structure**," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, Florida, pp. 22–30, October 1982.

[Reed and Kanodia78]

D. P. Reed and R. K. Kanodia, "**Synchronization with Eventcounts and Sequencers**," *Communications of the ACM*, 21:2, pp. 115–122, February 1978.

[Rushby84]

J. Rushby, "**The Security Model of Enhanced HDM**," *Proceedings of the 7th DOD/NBS Computer Security Conference*, pp. 120–136, September 1984.

[Rushby85]

J. Rushby, ***The SRI Security Model***, Technical Report, Computer Science Laboratory, SRI International, Menlo Park, California, April 1985.

[Saltzer and Schroeder75]

J. H. Saltzer and M. D. Schroeder, "**The Protection of Information in Computer Systems**," *Proceedings of the IEEE*, 63:9, pp. 1278–1308, September 1975.

[Schaefer77]

M. Schaefer, B. Gold, R. Linde, and J. Scheid, "**Program Confinement in KVM/370**," *Proceedings of the 1977 Annual ACM Conference*, Seattle, Washington, ACM, New York, pp. 404–410, October 1977.

[Schaefer89]

M. Schaefer, "**Symbol Security Condition Considered Harmful**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 20–46, April 1989.

[Schroeder77]

M. D. Schroeder, D. D. Clark, and J. H. Saltzer, "**The Multics Kernel Design Project**," *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, West Lafayette, IN, pp. 43–45, November 1977. (Also available in the *Operating Systems Review*, 11:5, November 1977.)

[Shannon and Weaver64]

C. E. Shannon and W. Weaver, ***The Mathematical Theory of Communication***, The University of Illinois Press, Urbana, Illinois, 1964.

[Shieh and Gligor90]

S. P. Shieh and V. D. Gligor, "**Auditing the Use of Covert Channels in Secure Systems**," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 1990.

[Tsai90]

C.-R. Tsai, V. D. Gligor, and C. S Chandersekaran, "**A Formal Method for the Identification of Covert Storage Channels in Source Code**," *IEEE Transactions on Software Engineering*, 16:6, pp. 569–580, June 1990. (Also in the *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 74–86, April 1987.)

[Tsai and Gligor88]

C.-R. Tsai and V. D. Gligor, "**A Bandwidth Computation Model for Covert Storage Channels and Its Applications**," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 108–121, April 1988.

[Walker83]

B. Walker, G. Popek, and G. Thiel, "**The LOCUS Distributed Operating System**," *Proceedings of the 9th ACM Symposium on Operating System Principles*, Bretton-Wood, New Hampshire, pp. 49–70, December 1983.

[Walter74]

K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, K. J. Biba, J. M. Gilligan, D. D. Schaeffer, S. I. Schaen, and D. G. Shumway, ***Modeling the Security Interface***, Technical Report No. 1158, Case Western Reserve University, Cleveland, Ohio, August 1974.

[Whitmore73]

J. Whitmore, et al., ***Design for Multics Security Enhancements***, Honeywell Information Systems Inc., Technical Report ESD-TR-74-176, HQ Electronic Systems Division, Hanscom AFB, Massachusetts, December 1973.

[Wray91]

J. C. Wray, "**An Analysis of Covert Timing Channels**," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, pp. 2–7, May 1991.

APPENDIX A
ADDITIONAL EXAMPLES OF COVERT CHANNELS

This appendix provides additional examples of storage and timing channels. For simplicity, in all covert channel examples below we assume the security level of the sending process S dominates that of the receiving process R. However, one can find similar examples where the security levels of S and R are incomparable using the dominance relation defined in the system.

A.1 STORAGE CHANNELS

The following examples of covert storage channels are identified in the literature; for example, see [Lopere85] or [Tsai and Gligor88]. These examples are necessarily generic in nature and are meant to be a starting point for identification of covert storage channels in specific systems.

A.1.1 Table-Space Exhaustion Channels

The table-space exhaustion channel is similar to the memory-resource exhaustion channel. The channel is present on systems in which the TCB allocates a fixed amount of space for its tables rather than employing some type of dynamic allocation scheme. For the purpose of creating an example here, suppose that a TCB has allocated a fixed table to keep track of processes created in the system. A sending process S whose security level dominates that of a receiving process R would transmit information in the following manner:

(1) Process R creates processes (one or more of these processes can be deleted by the sender S) until the process table space is exhausted. The new processes simply sleep or block indefinitely.

(2) Process R then synchronizes with process S.

(3) To send a 1 or 0, process S either deletes a process (in common with R) or doesn't and then blocks.

(4) Process R attempts to create another process (again, in common with S). It records a 1 or 0 depending on its success. Process R then synchronizes

with process S, and the operation continues in this fashion until all of the desired information is transferred.

A.1.2 Unmount of Busy File System Channels

This covert storage channel is exploitable in a segmented memory architecture system in which the file system cannot be unmounted if some segments are still in the address space of a process. If a process attempts to unmount a file system and is given an answer busy or not busy, a higher security level process owning segments contained in the file system can transfer information to a lower security level process attempting the unmount. The scenario is as follows:

(1) Process R (whose security level is dominated by that of S) begins by releasing all of the segments on the file system. Processes R and S have one segment in common which they can both map into their address spaces.

(2) Process R then synchronizes with process S.

(3) To send a 1 or 0, process S either maps the common segment into its address space (or does nothing if it is already there) or unmaps the common segment from its address space (or does nothing if it is already not there). Process S then blocks.

(4) Process R receives a 1 or 0 by attempting to unmount the file system and reviewing the result. If the unmount was successful, R remounts the file system.

(5) Process R then synchronizes with process S and the exchange continues until completion of the desired transfer.

A.1.3 Printer Attachment Channel

When physical printers or other I/O devices are shared resources in a system, a sending process S at a high security level could potentially transfer information to a receiving process R at a lower security level by creating contention for the the device(s). As always, the sender and receiver must have some way to synchronize. To send a 1, the sender process S simply checks to see if the printer is attached, and attaches it if it is not. To send a 0, the sender process S checks to see if the printer is attached, and detaches it if it is. The receiver process R attempts to attach the printer, receiving a 0 if successful, and a 1 otherwise. The receiver process R then detaches the printer if the attach call was successful.

A.2 TIMING CHANNELS

One way to think of the difference between covert timing channels and covert storage channels is that covert timing channels are essentially memoryless, whereas covert storage channels are not. With a timing channel, the information transmitted from the sender must be sensed by the receiver immediately, or it will be lost. However, an error code indicating a full disk which is exploited to create a storage channel may stay constant for an indefinite amount of time, so a receiving process is not as constrained by time.

As is the case with covert storage channels, covert timing channels will inevitably be present on any system in which sharing of system resources takes place. We present several examples of covert timing channels in the following sections.

The tasks of identification and handling covert timing channels (i.e., bandwidth reduction, elimination, or audit) in a secure system are more difficult than for covert storage channels for the following reasons:

(1) An accepted method (such as the SRM method presented in [Kemmerer83] and reviewed in Section 3.2.3) for identifying all covert timing channels does not exist. Although the SRM method has been presented as a tool for the identification of both covert storage and timing channels, in the case of timing channels it is no more effective than an ad hoc examination of each system call.

(2) Tools (such as Ina Flo or HDM MLS—presented in Appendix B) for identifying covert timing channels do not exist. This is largely because the formal specification methodologies that have been developed do not address timing issues, and this situation is not expected to change in the near future [Haigh87]. This is the case because none of the existent tools can help discover timing channel scenarios.

(3) Covert timing channels involve the exploitation of normal system activity (and sometimes the direct exploitation of hardware), thus they are much

more difficult to audit than covert storage channels. Attempts to perform meaningful audit of normal system activity will result in the generation of massive amounts of unusable data, added kernel complexity, and reduced performance. For example, it is practically impossible to audit the CPU timing channels of Example 3 and the bus, memory-port, and crossbar-switch contention channels of Example 4, Section 2.2.1.

Despite the problems listed above, to a certain extent one can design a secure system that limits or eliminates some types of covert timing channels. For example, one can eliminate a potential covert timing channel by time-partitioning a shared resource on a security-level basis (see Section 5.1). The addition of noise (for example, in the form of added processes) can also reduce the bandwidth of most covert timing channels.

The following sections enumerate some of the covert timing channels that are identified in the literature (for example, see [Schaefer77, Huskamp78, Karger and Wray91]). These examples are necessarily generic in nature and are meant to be a starting point for identification of covert timing channels in specific systems. We make two basic assumptions for all of the examples detailed in the following sections. First, in each scenario we assume the communicating processes either have a continuous clock available that has reasonable resolution, or the processes create a time reference by using simple counters in memory segments or files. Second, we assume the communicating processes are running alone on the system (i.e., little or no noise is present). This second assumption ensures calculating the maximum channel bandwidths.

These two assumptions help identify some of the countermeasures that can be used against some covert timing channels in a secure system. The first is to virtualize the clock in the system by resetting the clock at every context switch [Lipner75]. However, this action could render the system useless, since many system functions depend on a real, continuous time source. Also, this countermeasure is ineffective when the communicating processes have constructed their own time references. The other countermeasure that applies to most covert timing channels is the addition of noise to the system in the form of added processes. While this countermeasure can effectively reduce the bandwidth of the channel, it adds unwanted overhead to the system.

A.2.1 I/O Scheduling Channels

Consider a movable head I/O device that uses a scheduling algorithm called the "elevator algorithm." The algorithm works as follows: requests are enqueued by ascending cylinder number. Requests are then dequeued in order of ascending cylinder number until no greater cylinder number remains (i.e., the upper end of the cylinder is reached) and then are dequeued in descending order until no lower numbered cylinders remain (i.e., the lower end of the cylinder is reached). This process is continuously repeated.

Again, let process R be a receiver at a low security level and process S be the sender at a higher security level. Process R is the owner of cylinders 51 through 59 of a disk, to which S has read access. Process R issues a request for cylinder 55, waits until notified of its completion, and then relinquishes the CPU. Process S then issues a request for either cylinder 53 (to send a 0) or 57 (to send a 1), and then relinquishes the CPU. Process R then issues requests for cylinders 58 and 52, recording a 1 or 0 depending on which request completes first.

Note that similar timing channels can be found for other I/O scheduling algorithms. One way to eliminate these channels is to keep a process from viewing its requests until the entire queue of requests has been completed. This action also leads to underutilization of the I/O processing power.

A.2.2 I/O Operation Completion Channels

Assume process S and process R own separate portions of the same movable-head I/O device. Process S and process R synchronize themselves to run alternately by using the system's synchronization facilities. For an example of the synchronization primitives that could be used, see [Reed and Kanodia78]. To send a 1, process S requests a read on a part of the disk that is farthest from process R's portion of the disk. To send a 0, process S does nothing. Process R issues a read to its portion of the disk and determines the bit of information received depending on the time that it takes for the request to complete.

Note that both the I/O scheduling and the I/O completion channels are similar to the CPU quantum and CPU interquantum channels of Example 3, Section 2.2.1.

A.2.3 Memory Resource Management Channels

Below we describe two possible covert timing channels associated with the activity of memory resource management. Both involve a sending process conveying information by modulating the frequency with which a receiver process obtains a resource. The first channel involves the ability of the sender to modulate the data paging rate; the second involves the sender modulating the time the receiver takes to obtain addressability to a segment via an active segment table.

A.2.3.1 Data Page Pool Channels

Suppose a sender process S whose security level dominates that of a receiver process R can request the same page of data in a demand-paging environment. In this scenario, the receiver process R's security level must dominate the security level of the page, and the processes S and R must possess "read" access to the page. The sender process S may not possess "write" access to the page. Imagine also that the page has not been referenced, and so is not resident in main memory. Additionally, the processes S and R must agree on a set of pages to be used to exploit this covert channel. Process S can now modulate process R's response time (and thus send a bit of information to R) when reading the page by either referencing the page or not (bringing the page into memory or not). If process S is reading (or not reading) from a new page each time, it can continually send information to R. Since memory is not infinite, a new page cannot be referenced each time. A potentially continuous channel still exists, though, as long as process S has more pages available to it than the memory management working set size, the pages are referenced in a circular fashion, and the memory manager is using a Least Recently Used (LRU) page-replacement algorithm.

A.2.3.2 Active Segment Table Channels

The active segment table channel is very similar to the data page pool channel, and is only a threat in a segmented architecture. The sending process S either introduces a new segment into its address space (resulting in the segment being entered into the active segment table) or doesn't, depending on the desired value to be transmitted. The receiving process R introduces the same segment into its address space, and perceives the difference in response time, which varies depending upon whether or not the entry is already in the active segment table. Similar to the data page pool case, this channel can be made continuous as long as process S and process R share more segments than there are slots in the active segment table and process S releases the previous iteration's segment (removing it from the table) before proceeding to the next iteration.

A.2.4 Device Controller Contention Channels

Consider a system in which multiple single-level devices, of perhaps different levels, are handled by the same controller, and I/O to each device is scheduled serially. Process S, which is writing to device A, can send information to process R (R is dominated by S), which is writing to device B, by varying the time that it spends doing I/O. Note that this channel's bandwidth can potentially be raised by transmitting more elaborate bit patterns than just 0 or 1 with the use of encoding techniques (by associating higher radix digits with different time intervals perceived by the receiver). For example, a perceived interval of 0 to x would indicate a value of 0, interval x to y would be a 1, and interval y to z would be a 2. Note, however, that more elaborate schemes for encoding of data can be more susceptible to the effects of noise.

A.2.5 Exclusive Use of Segments Channels

In some systems, a user is allowed to obtain exclusive use of a segment. If a process currently has exclusive access to a segment, other requests for access to the segment are blocked until the segment becomes available. Therefore, a sending process S at a high security level could potentially gain exclusive access to a segment, and modulate the time it takes for a receiving process R at a lower security level to gain access to the segment. Note that, as in the previous section, the bandwidth of the channel could potentially be raised by the transmission of more elaborate bit patterns.

A.2.6 Synchronization Primitive Contention Channels

Consider a system where a central lock provides and controls process synchronization primitives. Contention for this central coordination can then be exploited to create a timing channel. The required configuration is similar to that of Example 4 of Section 2.2.1: three processors in a system, one process per processor, with a clock process incrementing a counter in a shared segment. The receiver process R continuously reads the shared counter, attempts two synchronization primitive calls, and reads the counter again. The sender process S (whose security level dominates that of the receiver process R) transmits a bit of data by either making a synchronization call (thereby causing contention) or waiting for the amount of time it would take to make two synchronization calls (one with contention), in order to stay synchronized. The receiver process R discriminates binary data by comparing the two reads of the counter in the shared segment. The delay will be longer if the sender process S has created contention.

APPENDIX B
TOOLS FOR COVERT CHANNEL IDENTIFICATION

The *TCSEC* requires one use formal methods in the covert channel analysis of a system targeted for the A1 class. A number of tools exist, and are generally associated with a particular suite of tools such as the Formal Development Methodology (FDM), the Enhanced Hierarchical Development Methodology (EHDM), and the Gypsy Verification Environment (GVE). Although the emphasis has been on examining specifications written in a language such as Ina Jo (FDM), Revised Special (EHDM), or Gypsy (GVE), some work has been done on analysis of source code for covert channels using tools (see Section 3.2). The examination for covert channels involves looking at each variable referenced in the specification, and deciding where information flow is possible.

The goal of using these tools is to identify (with respect to a given policy) insecure flows, so that all such flows in a system can be reduced or eliminated. However, as discussed in Section 3.1, use of a flow tool on a specification does not guarantee that insecure flows do not exist in an implementation; rather, it guarantees that insecure flows do not exist in, and are not required by, the specification. For this reason, one must be careful concerning assertions made about the application of a given tool to a design. Also, since the tools developed to date are not designed to find timing channel scenarios, they are useful primarily in the identification of covert storage channels.

The information-flow tools described below are the FDM Multilevel Secure (MLS) and SRM tools, the Gypsy Flow Analyzer, and the EHDM MLS tool. Chapter 3 describes the use of formal methods in the identification of covert channels in source code.

B.1 FDM INA FLOW TOOL

The Ina Flo Tool [Eckmann87], a software tool to aid CCA, is part of the FDM developed at Unisys. Ina Flo is composed of two tools: MLS, which is similar to the HDM MLS tool, and another tool, which implements the Shared Resource Matrix approach [Kemmerer83].

B.1.1 MLS

The MLS tool of Ina Flo identifies flows in an Ina Jo specification by examining dependencies between variables and formal parameters of transforms from one state to the next. If x and y are variables or formal parameters of a transform, and the new value of y depends on the old value of x, then information flows from x to y. MLS also includes the following rule for determining security: A flow is secure if and only if the label (security level) of variable y dominates the label of variable x (i.e., the *-property [Bell and La Padula76] is preserved). The user assigns labels to variables and defines a partial ordering on those labels (defines the dominates relation). MLS then generates a list of conjectures (one list per transform) which, when proven, guarantee there are no storage channels in the specification. Conjectures that cannot be proven represent potential covert channels which must be handled with manual analysis.

Because Ina Jo specifications can be written in a nondeterministic manner, the dependencies between old and new values of variables can be difficult to determine. For this reason, the generation of information regarding nondeterministic flows is optional. A preprocessor called "PREMLS" is available that accepts an Ina Jo specification and produces a more deterministic version of the same specification.

B.1.2 SRM

The SRM tool of Ina Flo is an implementation of the SRM Method, and is intended to be used on specifications in which the MLS tool cannot be used. This would be the case when the specification does not contain complete security policy information. The SRM tool does not provide as much automated capability as the MLS tool. It simply accepts the Ina Jo specification and generates the corresponding shared resource matrix. Analysis of this matrix is then a manual procedure.

B.2 GYPSY FLOW ANALYZER

The Gypsy Flow Analyzer [McHugh and Good85] is an information-flow tool that is part of the GVE. The basis for this tool comes from the Gypsy optimizer, which contains code to identify "ghost" variables having no effect on the outputs of the program. These variables are found by conducting a detailed flow analysis of the Gypsy specification. The flow analysis involves generating the set of all paths through each routine and determining all contributors to the output of the routine. In addition to the identification of flows resulting from assignment statements, those resulting from control constructs and buffer operations are identified.

The application of a flow tool such as this to the problem of identification of covert channels in secure systems involves the assignment of labels to variables, the definition of a flow policy, and the definition of a partial ordering among different label values, much like is done with the MLS tool of the FDM. Note that the choice of the flow policy is at the discretion of the user of the flow tool. The entire process of using the Gypsy Flow Tool is summarized in [McHugh and Good85] as follows:

(1) Definition of the desired information flow policy expressed as a Gypsy theory (i.e., a set of Gypsy functions, constants, lemmas, and data types).

(2) Identification of the TCB interface set.

(3) Execution of the flow analysis for the TCB.

(4) Construction of information flow analogues for each routine in the TCB interface set using the results of the previous step.

(5) Creation of information flow policy specifications for the TCB interface routines and for literals or constants appearing in the flows to the parameters of the interface set.

(6) Generation of verification conditions for the flow abstraction routines.

(7) Proof of the verification conditions.

Difficulty in proof of the verification conditions indicates existence of flows that violate the stated flow policy. The source of these illegal flows must be pinpointed exactly and treated so that the proof step can be completed. Unfortunately, in a large program, locating illegal flows can be very difficult, and the Gypsy Flow Tool does not provide much help in this area.

B.3 EHDM MLS TOOL

The EHDM MLS Tool [Rushby84] is an information flow tool that is part of the Enhanced Hierarchical Development Methodology. It accepts as input a specification written in Revised Special and produces a set of theorems to be verified. The successful verification of these theorems asserts that the specification is multilevel secure as defined by the SRI model in [Rushby84].

A conceptually simple statement of the SRI model is that the information users can obtain from a system cannot be influenced by users whose security level is greater than theirs. The model assumes a lattice of security levels and a collection of users assigned a security level. If operations in the system possess an invocation level SL_1, then the operation is multilevel secure if:

(1) The value (result) returned to the user depends on objects whose security levels SL_2 satisfy $SL_2 < = SL_1$.

(2) The objects that acquire new values during the operation are at security level SL_2 such that $SL_1 < = SL_2$.

(3) If an object at security level SL_2 acquires a new value dependent on the value of an object at security level SL_1, then SL_2 must dominate SL_1.

The SRI model is conceptually similar to the Bell and La Padula model [Bell and La Padula76]. The MLS tool produces a set of theorems for each operation that correspond to the three conditions stated above. Violations of the model may appear in the generated output, and may indicate design flaws or covert channels.

Two concerns about the MLS tool are raised in [Kemmerer86]. One is that the volume of generated theorems will be very high. The other is that all of the theorems may not be provable in an automated way using the EHDM theorem prover, so the user may have to edit the PROOF module, adding lemmas that make the verification conditions provable.

An earlier version of the MLS flow tool (old HDM) was used in the SCOMP verification effort [Benzel84] to identify a number of covert channels. Isolation of the potential channels consisted of tracing unprovable verification conditions back to the system specification, and then tracing the lines of the specification back to the lines of code, using the results of the specification-to-code correspondence effort. This result indicates it is highly desirable to complete the code correspondence effort before undertaking the task of applying the MLS flow tool to the specification.

B.4 SOURCE-CODE ANALYSIS TOOL

The previous three sections discuss various formal tools that can be used for the identification of covert storage channels by examining the specification of a system. This section outlines the steps of a formal method that has been developed for the identification of covert storage channels by examination of the source code of a system. [Tsai90, He and Gligor90] The advantages of a formal source-code approach to the identification of covert channels are:

(1) All potential storage channels in the implementation examined are discovered.

(2) It avoids discovery of false illegal flows (a problem that appears with the use of the other tools discussed above).

(3) The method helps to determine whether the mandatory access control rules are implemented correctly.

(4) It helps determine the source code locations where audit code, delays, and randomization code should be placed for handling covert channels.

(5) The method is a fully automated search for potential covert storage channels. [He and Gligor90]

In addition to the advantages stated above, a code level examination for covert storage channels seems to be stronger than the specification level searches provided by FDM, GVE, and HDM, since a formal method has not been developed for showing the correspondence of the specification to the code. The code-level search for covert storage channels is theoretically similar to specification-level approaches and is conducted as follows:

(1) Select the set of kernel primitives (TCB interface calls) to be analyzed.

(2) Determine the visibility/alterability of kernel variables when primitives are invoked.

(3) Apply the mandatory access control policy to the shared variables and kernel primitives to detect flows which are in violation.

For more details on this method of channel identification, we refer the reader to [Tsai90, He and Gligor90].

☆ U.S. GOVERNMENT PRINTING OFFICE:1994-517-749/80727

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
November 1993

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
A Guide to Understanding Covert Channel Analysis of Trusted Systems

5. FUNDING NUMBERS

6. AUTHOR(S)
Virgil Gligor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Security Agency
Attn: I94 (Standards, Criteria, and Guidelines Division)
Ft. George G. Meade, MD 20755-6000

8. PERFORMING ORGANIZATION REPORT NUMBER
NCSC-TG-030

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Security Agency
Attn: I94 (Standards, Criteria, and Guidelines Division)
Ft. George G. Meade, MD 20755-6000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Library No. S-240,572

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release: distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document provides a set of good practices related to covert channel analysis of systems employed for processing classified and other sensitive information. It's written to help vendors and evaluators understand covert channel analysis requirements. It contains suggestions and recommendations derived from Trusted Computer System Evaluation Criteria (TCSEC) objectives but which aren't required by the TCSEC.

14. SUBJECT TERMS
Computer security; Trusted Computer System Evaluation Criteria (TCSEC); automated information system (AIS); covert channel analysis; operating systems.

15. NUMBER OF PAGES
129

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT